Struts — The Complete Reference, Second Edition

Table of Contents

Chapter 1: An Introduction to Struts

Overview

Struts is the premier framework for building Java-based Web applications. Using the Model-View-Controller (MVC) design pattern, Struts solves many of the problems associated with developing high-performance, business-oriented Web applications that use Java servlets and JavaServer Pages. At the outset, it is important to understand that Struts is more than just a programming convenience. Struts has fundamentally reshaped the way that Web developers think about and structure a Web application. It is a technology that no Web developer can afford to ignore.

This chapter presents an overview of Struts, including the historical forces that drove its creation, the problems that it solves, and the importance of the Model-View-Controller architecture. The topics introduced here are examined in detail by subsequent chapters.

A Brief History of Web Application Development

In order to fully understand and appreciate the need for and value of Struts, it's necessary to shed some light on how Web application development has evolved over the past several years. Initially the Internet was used primarily by the academic and military communities for sharing research information, most of which was in the form of static documents. Thus, originally, the Internet was mostly a mechanism for sharing files.

In 1995 the commercialization of the Internet began and there was an explosion of content made available on the Web. Similar to the research content that was being shared on the Web, the early commercial content principally consisted of text mixed with simple graphics. Hyperlinks were used to connect the content together. Although hyperlinks enabled the user to move from page to page, each page was still a static document that did not support other forms of user interaction. It wasn't long, though, before businesses wanted to be able to offer dynamic content that offered the user a richer, more interactive experience.

Before continuing, it will be helpful to explain precisely what is meant by *dynamic content.* In short, dynamic content is data that is specifically targeted for a particular user. For example, a user may want to check the price and availability of some item in an online store. The user enters the item name in an HTML form and the server supplies the response. The response is generated on the fly based on the request, and is thus dynamic content.

To fill the dynamic-content void, Web server software began to support the use of CGI scripts for creating applications that could run on a Web server and generate dynamic content back to a browser. CGI, or *Common Gateway Interface,* allowed Web servers to accept a request and execute a server-side program that would perform some action and then generate output on standard out. Web server software would then read that output and send it back to the requesting browser. Initially, many of these CGI scripts were written in Perl or other Unix-based scripting languages. Over time, though, as the applications being built to run as CGI scripts grew in complexity, more application-oriented languages like C and C++ were being used to create larger, more robust applications. With the advent of HTML forms, CGI scripts also were able to receive data from the browser and process it. As most readers know, HTML forms allow data entry on a Web page. That data could be sent to a CGI script on the server and then manipulated, stored, or otherwise processed.

Around the same time that CGI-based application development was becoming popular on the server side, the Java programming language was introduced, with an initial focus on applets. Applets gave the Web developer the ability to add rich, dynamic functionality to Web pages. Because Java offered

the promise of "write once and run anywhere" programs, any browser that supported Java could run the applets. For the first time, developers could easily include dynamic content on a Web page.

For the same reasons that Java began to blossom on the client side with applets, Java also began to make inroads on the server side with the advent of servlet technology in 1997. Servlets solved many of the shortcomings of CGI, such as portability and efficiency, and offered a Java-based solution for the Web application paradigm. Servlets are portable across operating systems and can run on any server that has a Java Virtual Machine (JVM) and servlet container. Thus, they also benefit from Java's "write once, run anywhere" philosophy. Servlets have a more efficient execution model than CGIs because they are multithreaded instead of requiring a new process for each request. Servlets also have access to Java's vast libraries, including the JDBC APIs.

After servlets were introduced, Sun released the *JavaServer Pages (JSP)* technology as an extension to the servlet technology. JSPs take the reverse approach from servlets to building Web applications by having Java code intermingled in an HTML-based page. When a request is made to the server for a JSP, the Java servlet container checks if the JSP has already been compiled into a servlet. If it has, it proceeds to execute the servlet. If the JSP has not yet been compiled into a servlet, the server container converts the JSP code into a Java source file and then compiles that source so that subsequent requests to the JSP will find the servlet already compiled and ready to execute.

The nice thing about this approach is that changes to the JSP HTML can be made without having to manually recompile the code. The server container manages the compilation and will recognize that the HTML in the JSP has changed and recompile the JSP into a servlet for you. JSPs solve the problem of presentation code (HTML) being embedded in servlets, which made development cumbersome because HTML authors had to wade through Java code to edit HTML (not a good separation of responsibilities). In contrast, HTML developers can work on JSPs directly without interfering with Java code.

As the preceding discussion shows, many of the changes in Web-based development that have occurred over the past several years have been driven by the desire to efficiently include dynamic content in a Web page. Streamlining the use of dynamic content has been, and remains, one of the more important issues associated with the Internet and the applications that use it. As you will see, Struts is part of the solution to the dynamic-content problem.

Two Development Models

When Sun introduced JSP technology, it provided a development road map for working with it and defined two models for building JSP-based Web applications. The two models are known as*Model 1* and *Model 2* and they prescribe different approaches to designing JSP-based Web applications. Model 1, the simpler of the two, was the primary solution implemented when JSPs were first introduced. However, over time, Model 2 has been accepted as the best way for building JSP-based Web applications and, as you'll see, is the inspiration for MVC-based Web frameworks like Struts. Following is an overview of both architectures.

Model 1 Architecture Overview

The Model 1 architecture is very simple, as you can see in Figure 1-1. A request is made to a JSP or servlet and then that JSP or servlet handles all responsibilities for the request, including processing the request, validating data, handling the business logic, and generating a response. Although conceptually simple, this architecture is not conducive to large-scale application development because, inevitably, a great deal of functionality is duplicated in each JSP. Also, the Model 1 architecture unnecessarily ties together the business logic and presentation logic of the application. Combining business logic with presentation logic makes it hard to introduce a new "view" or access point in an application. For example, in addition to an HTML interface, you might want to include a

Wireless Markup Language (WML) interface for wireless access. In this case, using Model 1 will unnecessarily require the duplication of the business logic with each instance of the presentation code.

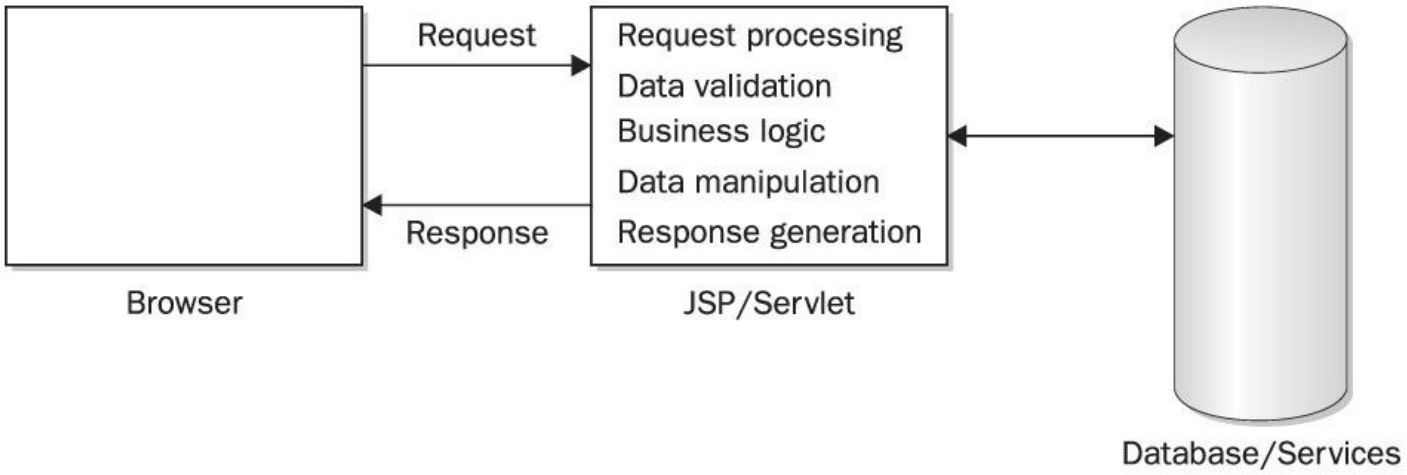


Figure 1-1: Model 1 architecture

Model 2 Architecture Overview

Model 2, or as it is most commonly referred to today, *Model-View-Controller (MVC),* solves many of the inherent problems with the original Model 1 design by providing a clear separation of application responsibilities (see Figure 1-2). In the MVC architecture, a central servlet, known as the *Controller,* receives all requests for the application. The Controller then processes the request and works with the *Model* to prepare any data needed by the *View* (which is usually a JSP) and forwards the data to a JSP. The JSP then uses the data prepared by the Controller to generate a response to the browser. In this architecture, the business and presentation logic are separated from each other. Having the separation of business and presentation code accommodates multiple interfaces to the application, be they Web, wireless, or GUI (Swing). Additionally, this separation provides excellent reuse of code.

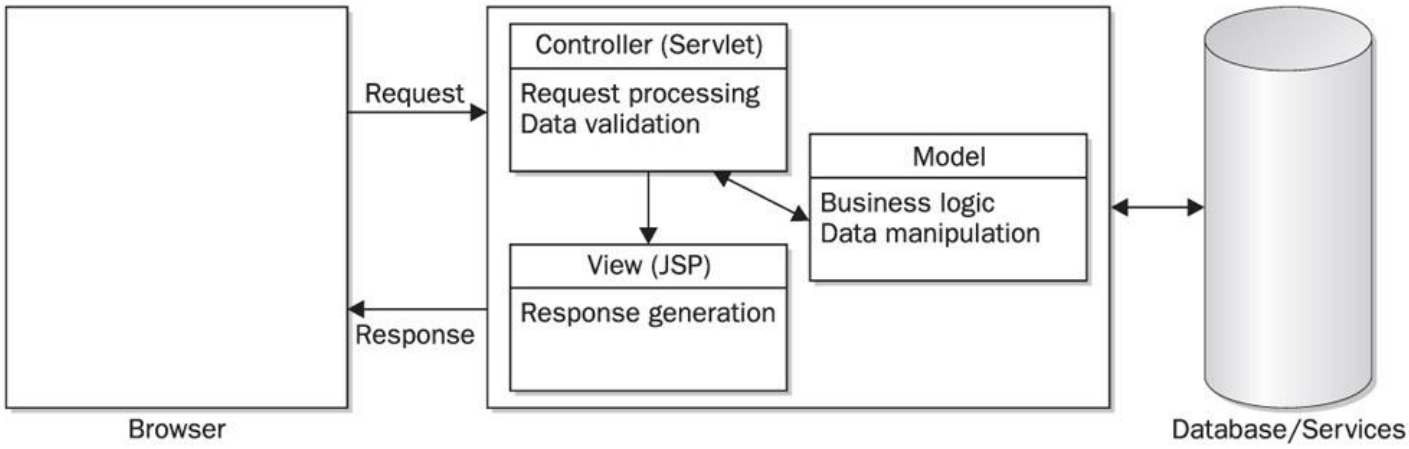


Figure 1-2: Model 2 architecture

A Closer Look at the Model-View-Controller Architecture

Because an understanding of the Model-View-Controller architecture is crucial to understanding Struts, this section takes a closer look at each of its parts. As a point of interest, MVC is based on an older graphical user interface (GUI) design pattern that has been around for some time, with its

origins in the Smalltalk world. Many of the same forces behind MVC for GUI development apply nicely to Web development.

Model Components

In the MVC architecture, model components provide an interface to the data and/or services used by an application. This way, controller components don't unnecessarily embed code for manipulating an application's data. Instead, they communicate with the model components that perform the data access and manipulation. Thus, the model component provides the business logic. Model components come in many different forms and can be as simple as a basic Java bean or as intricate as Enterprise JavaBeans (EJBs) or Web services.

View Components

View components are used in the MVC architecture to generate the response to the browser. Thus, a view component provides what the user sees. Oftentimes the view components are simple JSPs or HTML pages. However, you can just as easily use WML, a templating engine such as Velocity or FreeMarker, XML with XSLT, or another view technology altogether for this part of the architecture. This is one of the main design advantages of MVC. You can use any view technology that you'd like without impacting the Model (or business) layer of your application.

Controller Components

At the core of the MVC architecture are the controller components. The Controller is typically a servlet that receives requests for the application and manages the flow of data between the Model layer and the View layer. Thus, it controls the way that the Model and View layers interact. The Controller often uses helper classes for delegating control over specific requests or processes.

Enter Struts

Although the Model-View-Controller architecture is a powerful means of organizing code, developing such code can be a painstaking process. This is where Struts comes in. Struts is a Web application framework that streamlines the building of Web applications based on the MVC design principles. But what does that mean? Is Struts an MVC Web application that you just add on to or extend? Is Struts just some libraries? Actually, Struts is a little bit of both. Struts provides the foundation, or *framework,* for building an MVC-oriented application along with libraries and utilities for making MVC development faster and easier.

You could create a new Controller servlet every time you wanted to use the MVC design pattern in your Web application. Additionally, you'd need to create the management/flow logic for getting data to and from the Model and then routing requests to the View. You'd also need to define interfaces for interacting with your Model objects and all the utility code that goes along with using the MVC design pattern. However, instead of going through this process each time you create a new application, you can use Struts. Struts provides the basic structure and outline for building that application, freeing you to concentrate on building the business logic in the application and not the "plumbing."

To better understand the benefits of Struts, consider the following analogy. If you were to create a GUI application in Java, you wouldn't write a text field widget and a drop-down widget yourself. You would use Java's Swing API that already has standardized, fully functional code that provides these controls. Not only are the Swing controls ready-to-use, but they are also understood by all Java programmers. Struts provides the same type of advantages: Struts supplies a standard way of implementing an MVC application, the Struts code is tried and true, and the techniques required to use Struts are well known and documented.

In addition to providing the foundation for MVC applications, Struts provides rich extension points so that your application can be customized as you see fit. This extensibility has led to several third-party extensions being made available for Struts, such as libraries for handling application workflow, libraries for working with view technologies other than JSP, and so on. There is a brief overview of many of these popular third-party Struts extensions in Appendix B.

The Evolution of Struts

Struts was originally created by Craig R. McClanahan and then donated to the Jakarta project of the Apache Software Foundation (ASF) in May 2000. In June 2001, Struts 1.0 was released. Since then, many people have contributed both source code and documentation to the project and Struts has become the de facto standard for building Web applications in Java and has been embraced throughout the Java community. Struts 1.1 was released in June 2003 and included a substantial amount of new functionality, including the addition of Tiles, Validator, declarative exception handling, and much more. Later, in December 2004, Struts 1.2 was released with several minor updates to the framework. Struts 1.3, released in 2006, introduced the largest change to the framework since the 1.1 release: the move to a *chain of responsibility (COR)*-based request processing engine. The latest release of Struts at the time of this writing is 1.3.5, and that is the release covered in this book.

Today, Struts is continuing to evolve. There are now two distinct major versions of Struts: Struts 1 and Struts 2. Struts 1 is the mature, widely adopted, documented, and supported version of Struts. It is the version of Struts in use now and it is the version of Struts discussed in this book. Struts 2 is a completely new version of Struts based on the merger of Struts and WebWork, another popular open source Java Web application framework. At the time of this writing, Struts 2 is under development and does not have a specific release date. For this reason, Struts 2 is not discussed further in this book. Going forward, Struts 1 and Struts 2 will both be actively developed and maintained as separate subprojects of the Struts project.

Another event has punctuated Struts' evolution: its graduation to a top-level Apache project. Struts is no longer a subproject of the Jakarta Project. This is an important change because it affords Struts more autonomy and gives it the ability to have its own subprojects.

Struts Is Open Source

When Craig McClanahan donated Struts to the Apache Jakarta project, it became *open source software.* This means that anyone can download the source for Struts and modify that code as he or she sees fit. Of course, such changes affect only that developer. The standard code provided by ASF remains unaltered.

Over time, additional developers were added to the Struts project and were authorized to make changes to the code. These people are known as *committers,* since they have commit access to the source control repository for Struts. A limited number of people have this access, and each picks an area of interest and works on that part of the project that he or she is interested in.

A key advantage of open source is that it enables rapid development and maintenance cycles. For example, in an open source project, bugs can be fixed in a timely fashion. For ASF projects, bugs are handled by the committers, but anyone can fix a bug and provide a patch that the committers will then evaluate and "commit" if they deem it appropriate. Furthermore, anyone can contribute code to a project that incorporates changes and/or enhancements. Such submissions are also evaluated by the committers. In the case of Struts, contributions from members of the Struts community have played a significant role in the evolution of the project over the years.

Support for Struts comes in three forms. First is the API and usage documentation that comes with Struts. Second, Struts has a very active mailing list where you can get support for virtually any question. Third, several third-party consulting companies specialize in Struts support and development. Being open source, Struts is completely free of charge and allows you to make changes to it without any consequence so long as you abide by and preserve the ASF license.

Note Information for signing up to be a member of the Struts users mailing list can be found on the Struts Web site at http://struts.apache.org/.

Basic Components of Struts

The Struts framework is a rich collection of Java libraries and can be broken down into the following major pieces:

- Base framework
- JSP tag libraries
- Tiles plugin
- Validator plugin

A brief description of each follows.

Note In addition to the core pieces of the framework, the Struts project has an area known as the "Sandbox" where new code and ideas can be vetted. Code from the Sandbox is not packaged with Struts distributions but is available for download from the Sandbox section of the Struts Web site.

Base Framework

The base framework provides the core MVC functionality and consists of the building blocks for your application. At the foundation of the base framework is the Controller servlet: **ActionServlet**. The rest of the base framework is composed of base classes that your application will extend and several utility classes. Most prominent among the base classes are the **Action** and **ActionForm** classes. These two classes are used extensively in all Struts applications. **Action** classes are used by **ActionServlet** to process specific requests. **ActionForm** classes are used to capture data from HTML forms and to be a conduit of data back to the View layer for page generation.

JSP Tag Libraries

Struts comes packaged with several JSP tag libraries for assisting with programming the View logic in JSPs. JSP tag libraries enable JSP authors to use HTML-like tags to represent functionality that is defined by a Java class.

Following is a listing of the libraries and their purpose:

- **HTML** Used to generate HTML forms that interact with the Struts APIs.
- **Bean** Used to work with Java bean objects in JSPs, such as accessing bean values.
- **Logic** Used to cleanly implement simple conditional logic in JSPs.
- **Nested** Used to simplify access to arbitrary levels of nested objects from the HTML, Bean, and Logic tags.

Tiles Plugin

Struts comes packaged with the Tiles framework. Tiles is a rich JSP templating framework that facilitates the reuse of presentation (HTML) code. With Tiles, JSP pages can be broken up into individual "tiles" or pieces and then glued together to create one cohesive page. Similar to the design principles that the core Struts framework is built on, Tiles provides excellent reuse of View code. As of Struts 1.1, Tiles is part of and packaged with the core Struts download. Prior to Struts 1.1, Tiles was a

third-party extension but has since been contributed to the Struts project and is now more tightly integrated. Tiles has been well adopted in the Java Web development community and is being used outside of Struts with other Java Web frameworks. Because of this popularity, at the time of this writing an effort is underway to separate Tiles into its own project. This will allow Tiles to continue to evolve independently of Struts. Of course, Struts will still have seamless integration with Tiles.

Validator Plugin

Struts comes packaged, as of version 1.1, with the Jakarta Commons Validator framework for performing data validation. Validator provides a rich framework for performing data validation on both the server side and the client side (browser). Each validation is configured in an outside XML file so that validations can easily be added to and removed from an application declaratively versus being hard-coded into the application. Similar to Tiles, prior to Struts 1.1, Validator was a third-party extension, but it has since been included in the project and is more tightly integrated.

Acquiring Struts

Struts is available free of charge and can be downloaded from the Apache Struts site at http://struts.apache.org/. Because Struts is open source, you have a couple of options when downloading the Struts framework software. You can download the software in binary, precompiled form or you can download the source code for compiling on your own. For most cases, the binary distribution will suffice; however, if you want to make changes to the Struts source code, the source distribution is available.

If you choose to download a binary distribution of Struts, you have a couple of options. You can download a released version of the code, which has been rigorously tested and certified as being of good quality, or you can download a nightly build of the code, which is less stable and not intended for production use. Opting to use a nightly build allows you to get access to the latest enhancements and bug fixes that have been made to the Struts framework ahead of an official release. However, it's important to point out again that nightly builds have no guarantee on quality because adding a new feature to Struts could potentially break another feature that has been stable for some time.

Similar to downloading a binary distribution of Struts, if you choose to download a source distribution, you have a couple of options. You can download the source for an officially released version of Struts or you can choose to get the "latest and greatest" version of the Struts source code directly from the Struts Subversion source control repository. Just as with the binary distribution, choosing to download the latest Struts source code can get you the newest enhancements and bug fixes to the software, but it may also have new bugs.

What You Get (Binary)

Since Struts is a Web application framework and not a stand-alone application, Struts distributions principally consist of the Struts API libraries and their associated files, such as Document Type Definitions (DTDs) for XML configuration files and JSP Tag Library Descriptor (TLD) files. Additionally, Struts comes with several sample Web applications that illustrate how to use the Struts framework. One of the sample Web applications, **struts-blank-[strutsversion #].war** (e.g., **struts-blank-1.3.5.war**), is typically used for new Struts applications because it provides a basic template for a Struts application, including all the necessary **.jar** files, and so on. Struts distributions also come with a sample Web application, **struts-mailreader-[struts version #].war**(e.g., **struts-mailreader-1.3.5.war**), that illustrates the basic structure of a Struts application.

What You Get (Source)

Similar to the binary distribution, the source distribution consists of the Struts API libraries and sample Web applications. The major difference, however, is that all of the code for the libraries and sample applications is in source form. This is particularly useful for projects where the source code may need to be changed or where you may want access to the source code for debugging an application and so on.

Getting Started with Struts

Now that the theoretical foundation for Struts has been covered, it is time to move on to actually writing Struts code. The next chapter walks through an example Struts application. Before then, you will need to choose one of the two Struts distribution options just discussed and download it at: http://struts.apache.org/downloads.html.

Chapter 2: Building a Simple Struts Application

Overview

Now that you've reviewed the history of Web application development and the fundamentals of the Struts framework, it's time to move beyond theory and into practice. As you will see, a Struts application is a composite of several interrelated parts. The goal of this chapter is to give you a general understanding of these parts and show how they work together to form a complete program. To accomplish that goal, this chapter develops a simple application that highlights each component of a Struts application. In the process, several key elements of Struts are introduced. Once you understand how this simple Struts application works, you will be able to easily understand other Struts programs because all share a common architecture. Subsequent chapters discuss in detail the many concepts introduced here.

Application Overview

The sample application in this chapter deviates from the stereotypical "Hello World" program found in many programming books. Instead, a bit more sophisticated example is needed to illustrate the components of Struts and the process required to build a Struts-based application. The example that we will use is a simple human resources (HR) application called Mini HR. Creating a full-blown HR application is a large undertaking that requires several pieces of functionality, from employee management to benefits management, so the sample application in this chapter will support only one common subset of functionality: Employee Search.

The Mini HR Application Files

All Struts applications consist of several files, which contain the various parts of a Struts program. Some are Java source files, but others contain JSP and XML. A resource bundle properties file is also required. Because of the relatively large number of files required by a Struts application, we will begin by examining the files required by Mini HR. The same general types of files will be needed by just about any Struts application.

The following table lists each file required by Mini HR and its purpose.

File	Description
index.jsp	Contains the JSP that is used as a gateway page for the Mini HR application and provides a link to the Employee Search page.
search.jsp	Contains the JSP that is used for performing employee searches and displaying the search results.
SearchForm.java	Contains the class that captures and transfers data to and from the Search page. This is a View class.
SearchAction.java	Contains the class code that processes requests from the Search page. This is a Controller class.
EmployeeSearchService.java	Contains the class that encapsulates the business logic and data access involved in searching for employees. This is a Model class.
Employee.java	Contains the class that represents an employee and encapsulates all of an employee's data. This is a Model class.

File	Description
web.xml	Contains the XML that is used to configure the servlet container properties for the Mini HR Java Web application.
struts-config.xml	Contains the XML that is used to configure the Struts framework for this application.
MessageResources.properties	Contains properties that are used to externalize application strings, labels, and messages so that they can be changed without having to recompile the application. This file is also used for internationalizing the application.

The following sections examine each of the Mini HR application files in detail, and in many cases line by line. First, though, it's necessary to explain where each file should be placed

in a directory hierarchy. Because this application (and all other Struts applications) will be deployed to a servlet container, the application files have to be arranged in the standard Web Archive (**.war**) format, which is simply a Java Archive (**.jar**) file with a different extension (**.war**). The Web Archive format also specifies a few key requirements for the **.jar** file:

- There must be a directory at the root level of the archive named **WEB-INF**. At run time this is a protected directory, and thus any files beneath it will be inaccessible to direct access by browsers.
- There must be a Web application deployment descriptor file named **web.xml** beneath the **WEB-INF** directory. This file will be explained later in this chapter, in the section "web.xml".
- Any libraries (**.jar** files) needed by the application should be under a directory called **lib** located beneath the **WEB-INF** directory.
- Any class files or resources needed by the application, which are not already packaged in a **.jar** file, should be under a directory called **classes** located beneath the **WEB-INF**directory.

For the Mini HR application, you will create a directory called **MiniHR**. In principle, you can place this directory anywhere, but to follow along with this example, put it at **c:\java**. You'll use the**c:\java\MiniHR** directory as the root of your Web application so that you can easily create a Web Archive file later. Following is the layout of the **c:\java\ MiniHR** directory, shown in Figure 2-1, and the location of each file examined in this section. You will need to place the files in this exact structure.

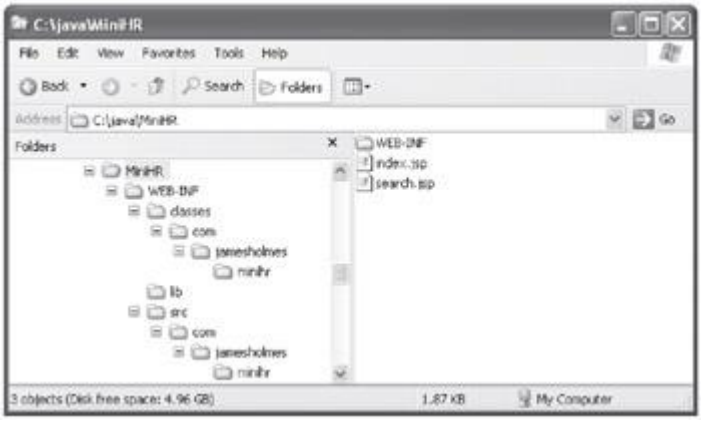


Figure 2-1: The c:\java\MiniHR directory layout

```
c:\java\MiniHR\index.jsp
c:\java\MiniHR\search.jsp
c:\java\MiniHR\WEB-INF\web.xml
```

```
c:\java\MiniHR\WEB-INF\struts-config.xml
c:\java\MiniHR\WEB-
INF\classes\com\jamesholmes\minihr\MessageResources.properties
c:\java\MiniHR\WEB-INF\lib
c:\java\MiniHR\WEB-INF\src\com\jamesholmes\minihr\Employee.java
c:\java\MiniHR\WEB-
INF\src\com\jamesholmes\minihr\EmployeeSearchService.java
c:\java\MiniHR\WEB-INF\src\com\jamesholmes\minihr\SearchAction.java
c:\java\MiniHR\WEB-INF\src\com\jamesholmes\minihr\SearchForm.java
```

index.jsp

The **index.jsp** file, shown here, is a very simple JSP that is used to render Mini HR's opening screen:

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>

<html>
<head>
<title>ABC, Inc. Human Resources Portal</title>
</head>
<body>

<font size="+1">ABC, Inc. Human Resources Portal</font><br>
<hr width="100%" noshade="true">

&#149; Add an Employee<br>
&#149; <html:link forward="search">Search for Employees</html:link><br>

</body>
</html>
```

You'll notice that **index.jsp** consists mostly of standard HTML, with the exception of the JSP tag library definition at the top of the file and the "Search for Employees" link. The **index.jsp** file uses the Struts HTML Tag Library to render the Search link. Before you can use the HTML Tag Library, you have to "import" it into the JSP with the following line at the top of the JSP:

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
```

This line associates the tag library descriptor with a URI of **http://struts.apache.org/tagshtml** with a prefix of "html". That way, any time a tag from the Struts HTML Tag Library is used, it will be prefixed with "html". In **index.jsp**'s case, the **link** tag is used with the following line:

```
<html:link forward="search">Search for Employees</html:link>
```

Of course, if you wanted to use another prefix for the tag library, you could do so by updating the **prefix** attribute of the tag library import on the first line of the file.

The HTML Tag Library's **link** tag is used for rendering an HTML link, such as http://www.jamesholmes.com/. The **link** tag goes beyond basic HTML, though, by allowing you to access link, or*forward* definitions (Struts terminology), from the Struts configuration file (e.g., **struts-config.xml**), which is covered later in this chapter, in the section "struts-config.xml". In this case, the tag looks for a forward definition named "search" defined in the **struts-config.xml** file to use for the link being generated. If you skip ahead to the "strutsconfig.xml" section of this chapter, you'll see that the forward definition is as follows:

```
<!-- Global Forwards Configuration -->
<global-forwards>
  <forward name="search" path="/search.jsp"/>
</global-forwards>
```

Forward definitions allow you to declaratively configure the location to which a link points instead of hard-coding that information into your JSP or application. As you'll see in Chapter 5, forward definitions are used throughout Struts to direct the flow of an application from the Struts configuration file.

The following is the source code generated after **index.jsp** has been requested in the browser. Notice that the Search page link has been converted into a standard HTML link.

```
<html>
<head>
<title>ABC, Inc. Human Resources Portal</title>
</head>
<body>
<font size="+1">ABC, Inc. Human Resources Portal</font><br>

<hr width="100%" noshade="true">

&#149; Add an Employee<br>
&#149; <a href="/MiniHR/search.jsp">Search for Employees</a><br>

</body>
</html>
```

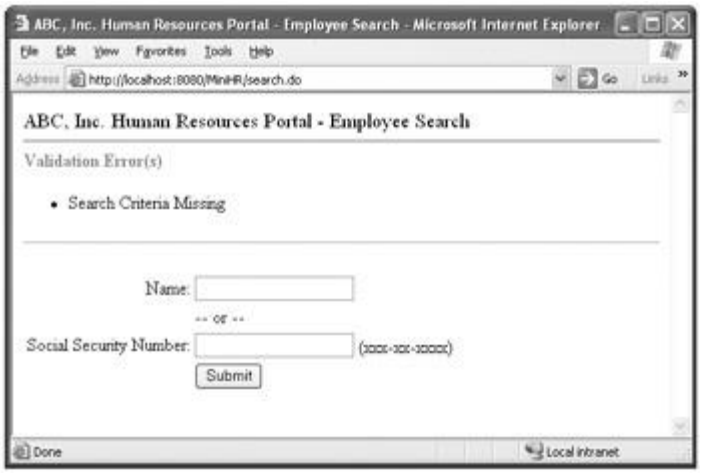
Here is how **index.jsp** looks in the browser.

search.jsp

The **search.jsp** file is responsible for the bulk of the Employee Search functionality in the Mini HR application. When the Employee Search link is selected from the **index.jsp** page, **search.jsp**is executed. This initial request for **search.jsp** renders the basic Employee Search screen shown here:

Each time a search is performed, the Struts Controller Servlet is executed and eventually **search.jsp** is executed to handle the rendering of the Employee Search screen, with the search results, as shown here:

Similarly, if there are any errors with the search criteria when the search is submitted, **search.jsp** is executed to report the errors, as shown here:



The contents of **search.jsp** are

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>

<html>
<head>
<title>ABC, Inc. Human Resources Portal - Employee Search</title>
</head>
<body>

<font size="+1">
ABC, Inc. Human Resources Portal - Employee Search
</font><br>
<hr width="100%" noshade="true">

<html:errors/>

<html:form action="/search">

<table>
<tr>
<td align="right"><bean:message key="label.search.name"/>:</td>
<td><html:text property="name"/></td>
</tr>
<tr>
<td></td>
<td>-- or --</td>
</tr>
```

```
<tr>
<td align="right"><bean:message key="label.search.ssNum"/>:</td>
<td><html:text property="ssNum"/> (xxx-xx-xxxx)</td>
</tr>
<tr>
<td></td>
<td><html:submit/></td>
</tr>
</table>

</html:form>

<logic:present name="searchForm" property="results">

<hr width="100%" size="1" noshade="true">

<bean:size id="size" name="searchForm" property="results"/>
<logic:equal name="size" value="0">
<center><font color="red"><cTypeface:Bold>No Employees
Found</b></font></center>
</logic:equal>

<logic:greaterThan name="size" value="0">
<table border="1">
<tr>
<th>Name</th>
<th>Social Security Number</th>
</tr>
<logic:iterate id="result" name="searchForm" property="results">
<tr>
<td><bean:write name="result" property="name"/></td>
<td><bean:write name="result" property="ssNum"/></td>
</tr>
</logic:iterate>
</table>
</logic:greaterThan>

</logic:present>
```

```
</body>
</html>
```

Because of its size and importance, we will examine it closely, line by line.

Similar to **index.jsp**, **search.jsp** begins by declaring the JSP tag libraries that will be used by the JSP:

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>
```

In addition to the HTML Tag Library used by **index.jsp**, **search.jsp** uses the Struts Bean and Logic libraries. These additional tag libraries contain utility tags for working with Java beans and using conditional logic in a page, respectively.

The next several lines consist of basic HTML tags:

```
<html>
<head>
<title>ABC, Inc. Human Resources Portal - Employee Search</title>
</head>
<body>

<font size="+1">
ABC, Inc. Human Resources Portal - Employee Search
</font><br>
<hr width="100%" noshade="true">
```

Immediately following this basic HTML is this **errors** tag definition:

```
<html:errors/>
```

Recall that **search.jsp** is used to render any errors that occur while validating that the search criteria are sound. The HTML Tag Library's **errors** tag will emit any errors that are passed to the JSP from the **SearchForm** object. This is covered in more detail in the section "SearchForm.java" in this chapter.

The next several lines of **search.jsp** are responsible for rendering the HTML for the search form:

```
<html:form action="/search">

<table>
<tr>
<td align="right"><bean:message key="label.search.name"/>:</td>
<td><html:text property="name"/></td>
</tr>
<tr>
<td></td>
```

```
<td>-- or --</td>
</tr>
<tr>
<td align="right"><bean:message key="label.search.ssNum"/>:</td>
<td><html:text property="ssNum"/> (xxx-xx-xxxx)</td>
</tr>
<tr>
<td></td>
<td><html:submit/></td>
</tr>
</table>

</html:form>
```

Before discussing the specifics of the search form, let's review the use of the Bean Tag Library in this snippet. This snippet uses the library's **message** tag, as shown here:

```
<td align="right"><bean:message key="label.search.name"/>:</td>
```

The **message** tag allows externalized messages from the **MessageResources.properties** resource bundle file to be inserted into the JSP at run time. The **message** tag simply looks up the key passed to it in **MessageResources.properties** and returns the corresponding message from the file. This feature is especially useful to internationalize a page and to allow easy updating of messages outside the JSP. *Internationalization* is the process of providing content specific to a language, locale, or region. For instance, internationalization would be to create both English and Spanish versions of the same JSP.

Note The acronym I18N is sometimes used in place of the word internationalization, because it is such a long word to type. I18N represents the first letter i, followed by 18 characters, and then the final letter n.

Now, it's time to examine the form. The Struts HTML Tag Library has a tag for each of the standard HTML form tags, such as
<form>

<input type="text">

<input type="checkbox">

<input type="radio">

<select>

and so on. Instead of using the standard HTML tags, you'll use the HTML Tag Library's equivalent tag, which ties the form to Struts. For example, the **text** tag (**<html:text>**) renders an **<input type="text" ...>** tag. The **text** tag goes one step further, though, by allowing a property to be associated with the tag, as shown here:

```
<td><html:text property="name"/></td>
```

The property "name" here corresponds to the field named **name** in the **SearchForm** object. That way, when the tag is executed, it places the value of the **name** field in the HTML at run time. Thus, if

the **name** field had a value of "James Holmes" at run time, the output from the tag would look like this:

```
<td><input type="text" name="name" value="James Holmes"></td>
```

At the beginning of this snippet, the HTML Tag Library's **form** tag is used to render a standard HTML **<form>** tag. Notice, however, that it specifies an **action** parameter of "/ search" as shown here:

```
<html:form action="/search">
```

The **action** parameter associates an **Action** object mapping from the Struts configuration file with the form. That way, when the form is submitted, the processing will be handled by the specified **Action** object.

The final section of the **search.jsp** file contains the logic and tags for rendering search results:

```
<logic:present name="searchForm" property="results">


<hr width="100%" size="1" noshade="true">


<bean:size id="size" name="searchForm" property="results"/>
<logic:equal name="size" value="0">
<center><font color="red"><cTypeface:Bold>No Employees
Found</b></font></center>
</logic:equal>


<logic:greaterThan name="size" value="0">
<table border="1">
<tr>
<th>Name</th>
<th>Social Security Number</th>
</tr>
<logic:iterate id="result" name="searchForm" property="results">
<tr>
<td><bean:write name="result" property="name"/></td>
<td><bean:write name="result" property="ssNum"/></td>
</tr>
</logic:iterate>
</table>
</logic:greaterThan>


</logic:present>
```

The beginning of this snippet uses the Struts Logic Tag Library for implementing conditional logic in a JSP. The Logic Library's **present** tag checks an object to see if a particular property is present. In this case, the **logic** tag checks to see if the **results** field of the **SearchForm** has been set. If so, then all

of the HTML and JSP tags inside the **<logic:present>** tag will be executed. Otherwise, they will be ignored.

The rest of the tags in this snippet are responsible for rendering the search results. First, the Bean Library's **size** tag gets the size of the **results ArrayList** from the **SearchForm** object. Next, the size is checked to see if it is 0 using the Logic Library's **equal** tag. If the size is 0, then a "No Employees Found" message will be rendered. Otherwise, each of the employees returned from the search will be displayed. The Logic Library's **iterate** tag is used to iterate over each of the search results. Each search result is assigned to a variable named **result** by the **iterate** tag. Inside the **iterate** tag the Bean Library's **write** tag is used to access the **result** variable's **name** and **ssNum** fields.

SearchForm.java

The **SearchForm** class, shown next, is a View class that is used to capture and transfer data to and from the Employee Search page. When the HTML form on the Search page is submitted, the Struts **ActionServlet** will populate this class with the data from the form. Notice that there will be a one-to-one mapping between fields on the page and fields in the class with getter and setter methods. Struts uses encapsulation and Java's reflection mechanism to call the method corresponding to each field from a page. Additionally, when **SearchAction** (the Controller class for the Search page) executes, it will populate this object with the search results so that they can be transferred back to the Search page.

```
package com.jamesholmes.minihr;

import java.util.List;

import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;

public class SearchForm extends ActionForm
{
  private String name = null;
  private String ssNum = null;
  private List results = null;

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
```

```
}

public void setSsNum(String ssNum) {
  this.ssNum = ssNum;
}

public String getSsNum() {
  return ssNum;
}

public void setResults(List results) {
  this.results = results;
}

public List getResults() {
  return results;
}

// Reset form fields.

public void reset(ActionMapping mapping, HttpServletRequest request)
{
  name = null;
  ssNum = null;
   results = null;
}

// Validate form data.
public ActionErrors validate(ActionMapping mapping,
  HttpServletRequest request)
{
  ActionErrors errors = new ActionErrors();

  boolean nameEntered = false;
  boolean ssNumEntered = false;

  // Determine if name has been entered.
  if (name != null && name.length() > 0) {
      nameEntered = true;
```

```
  }

      // Determine if social security number has been entered.
  if (ssNum != null && ssNum.length() > 0) {
    ssNumEntered = true;
  }

  /* Validate that either name or social security number
     has been entered. */
  if (!nameEntered && !ssNumEntered) {
    errors.add(null,
          new ActionMessage("error.search.criteria.missing"));
  }

  /* Validate format of social security number if
       it has been entered. */
  if (ssNumEntered && !isValidSsNum(ssNum.trim())) {
      errors.add("ssNum",
      new ActionMessage("error.search.ssNum.invalid"));
  }

  return errors;
}

// Validate format of social security number.
private static boolean isValidSsNum(String ssNum) {
    if (ssNum.length() < 11) {
     return false;
  }

  for (int i = 0; i < 11; i++) {
    if (i == 3 || i == 6) {
      if (ssNum.charAt(i) != '-') {
        return false;
      }
    } else if ("0123456789".indexOf(ssNum.charAt(i)) == -1) {
      return false;
    }
  }
```

```
    return true;
  }
}
```

ActionForm subclasses, including **SearchForm**, are basic Java beans with a couple of extra Struts-specific methods: **reset**( ) and **validate**( ). The **reset**( ) method is used to clear out, or "reset," an **ActionForm**'s data after it has been used for a request. When using session-based Form Beans, Struts reuses **ActionForm** instances instead of creating new ones for each request. The **reset( )** method is necessary to ensure that data from different requests is not mixed. Typically, this method is used to just set class fields back to their initial states, as is the case with **SearchForm**. However, as you'll see in Chapter 4, this method can be used to perform other necessary logic for resetting an **ActionForm** object.

The **validate( )** method of **ActionForm** is called to perform basic validations on the data being transferred from an HTML form. In **SearchForm**'s case, the **validate( )** method first confirms that a name and a social security number have been entered. If a social security number has been entered, **SearchForm** goes one step further and validates the format of the social security number with the **isValidSsNum( )** method. The **isValidSsNum( )** method simply ensures that an 11-character string was entered and that it conforms to the following format: three digits, hyphen, two digits, hyphen, four digits (e.g., 111-22-3333). Note that business-level validations, such as looking up a social security number in a database to make sure it is valid, are considered business logic and should be in a Model-layer class. The validations in an **ActionForm** are meant to be very basic, such as just confirming that data was entered, and should not be used for performing any real business logic.

You'll notice that the **validate( )** method returns an **ActionErrors** object and the validations inside the method populate an **ActionErrors** object if any validations fail. The **ActionErrors** object is used to transfer validation error messages to the screen. Remember from the discussion of **search.jsp** that the HTML Tag Library's **errors** tag will emit any errors in a JSP if they are present. Following is the snippet from **search.jsp**:

```
<html:errors/>
```

Here in the **ActionForm** class, you simply place the keys for messages into the **ActionErrors** object, such as "error.search.criteria.missing". The **errors** tag will use these keys to load the appropriate messages from the **MessageResources.properties** resource bundle file, discussed in the section of the same name later in this chapter.

SearchAction.java

The **SearchAction** class, shown next, is a Controller class that processes requests from the Search page:

```
package com.jamesholmes.minihr;

import java.util.ArrayList;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
```

```
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public final class SearchAction extends Action
{
  public ActionForward execute(ActionMapping mapping,
     ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
     throws Exception
  {
    EmployeeSearchService service = new EmployeeSearchService();
    ArrayList results;

    SearchForm searchForm = (SearchForm) form;

    // Perform employee search based on the criteria entered.
    String name = searchForm.getName();
       if (name != null && name.trim().length() > 0) {
      results = service.searchByName(name);
    } else {
      results = service.searchBySsNum(searchForm.getSsNum().trim());
    }

    // Place search results in SearchForm for access by JSP.
    searchForm.setResults(results);

       // Forward control to this Action's input page.
    return mapping.getInputForward();
  }
}
```

Remember from the discussion of **search.jsp** that the HTML form on the page is set to post its data to the "/search" action. The **strut-config.xml** file maps the search action to this class so that when **ActionServlet** (Controller) receives a post from the Search page, it delegates processing for the post to this **Action** subclass. This mapping is shown here:

```
<!-- Action Mappings Configuration -->
<action-mappings>
  <action path="/search"
```

```
            type="com.jamesholmes.minihr.SearchAction"
            name="searchForm"
            scope="request"
            validate="true"
            input="/search.jsp">
    </action>
</action-mappings>
```

Struts **Action** subclasses manage the processing of specific requests. You can think of them as mini-servlets assigned to manage discrete Controller tasks. For instance, in the preceding example, **SearchAction** is responsible for processing employee search requests and acts as a liaison between the Model (**EmployeeSearchService**) and the View (**search.jsp**).

SearchAction begins by overriding the **Action** class' **execute( )** method. The **execute( )** method is the single point of entry for an **Action** class by the Struts **ActionServlet**. You'll notice that this method takes an **HttpServletRequest** object and an **HttpServletResponse** object as parameters, similar to a servlet's **service( )**, **doGet( )**, and **doPost( )** methods. Additionally,**execute( )** takes a reference to the **ActionForm** associated with this **Action** and an **ActionMapping** object reference. The **ActionForm** reference passed to this **Action** will be an instance of**SearchForm**, as discussed in the preceding section, "SearchForm.java." The **ActionMapping** reference passed to this **Action** will contain all of the configuration settings from the **struts-config.xml** file for this **Action**.

The **execute( )** method begins by instantiating a few objects, and then the real work gets underway with a check to see what search criteria were entered by the user. Notice that the**ActionForm** object passed in is cast to its native type: **SearchForm**. Casting the object allows the **SearchForm** methods to be accessed for retrieving the search criteria. Based on the criteria entered, one of the **EmployeeSearchService** methods will be invoked to perform the employee search. If an employee name was entered, the **searchByName( )** method will be invoked. Otherwise, the **searchBySsNum( )** method will be invoked. Both search methods return an **ArrayList** containing the search results. This **results ArrayList** is then added to the**SearchForm** instance so that **search.jsp** (View) can access the data.

The **execute( )** method concludes by forwarding control to the **SearchAction** input page: **search.jsp**. The input page for an action is declared in the Struts configuration file, as shown here for**SearchAction**, and is used to allow an action to determine from which page it was called:

```
<action path="/search"
        type="com.jamesholmes.minihr.SearchAction"
        name="searchForm"
        scope="request"
        validate="true"
        input="/search.jsp">
```

EmployeeSearchService.java

EmployeeSearchService is a Model class that encapsulates the business logic and data access routines involved in searching for employees. The **SearchAction** Controller class uses this class to perform an employee search and then shuttles the resulting data to the View layer of the Mini HR application. **EmployeeSearchService** is shown here:

```
package com.jamesholmes.minihr;

import java.util.ArrayList;

public class EmployeeSearchService
{
  /* Hard-coded sample data. Normally this would come from a real data
     source such as a database. */
  private static Employee[] employees =
  {
    new Employee("Bob Davidson", "123-45-6789"),
    new Employee("Mary Williams", "987-65-4321"),
    new Employee("Jim Smith", "111-11-1111"),
    new Employee("Beverly Harris", "222-22-2222"),
    new Employee("Thomas Frank", "333-33-3333"),
    new Employee("Jim Davidson", "444-44-4444")
  };

  // Search for employees by name.
  public ArrayList searchByName(String name) {
       ArrayList resultList = new ArrayList();

    for (int i = 0; i < employees.length; i++) {
      if(employees[i].getName().toUpperCase().indexOf(name.toUpperCase())
         != -1)
      {
        resultList.add(employees[i]);
      }
    }

    return resultList;
  }

  // Search for employee by social security number.
  public ArrayList searchBySsNum(String ssNum) {
    ArrayList resultList = new ArrayList();

    for (int i = 0; i < employees.length; i++) {
      if (employees[i].getSsNum().equals(ssNum)) {
```

```
        resultList.add(employees[i]);
      }
    }

    return resultList;
  }
}
```

In order to simplify Mini HR, the **EmployeeSearchService** class will not actually communicate with a real data source, such as a database, to query employee data.
Instead,**EmployeeSearchService** has some sample employee data hard-coded at the top of the class, as shown here:

```
/* Hard-coded sample data. Normally this would come from a real data
   source such as a database. */
private static Employee[] employees =
{
  new Employee("Bob Davidson", "123-45-6789"),
  new Employee("Mary Williams", "987-65-4321"),
  new Employee("Jim Smith", "111-11-1111"),
  new Employee("Beverly Harris", "222-22-2222"),
  new Employee("Thomas Frank", "333-33-3333"),
  new Employee("Jim Davidson", "444-44-4444")
};
```

The sample data consists of a few **Employee** objects. As you'll see in the next section, the **Employee** class is a simple class for encapsulating employee data.

The **searchByName( )** and **searchBySsNum( )** methods use the hard-coded data when performing a search. The **searchByName( )** method loops through each of the **Employee** objects in the **employees** array looking for any employees that match the name specified. If a match is found, it is added to the return **ArrayList** that will eventually be used by **search.jsp** to display the results. Note that the name search is case insensitive by virtue of uppercasing the **String**s before comparison. You should also note that the use of **String**'s **indexOf( )** method allows for partial matches instead of only exact matches.

Similar to the **searchByName( )** method, **searchBySsNum( )** loops through the hard-coded employee list looking for any employees that match the specified social security number. Note that**searchBySsNum( )** will capture only exact matches. Because social security numbers are unique to an individual, only one match should ever be returned for a social security number–based search.

Employee.java

The **Employee** class, shown next, is a basic class for encapsulating the data for an employee. The class is straightforward, consisting simply of setters and getters for the **Employee** class data.

```
package com.jamesholmes.minihr;
```

```
public class Employee
{
  private String name;
  private String ssNum;

  public Employee(String name, String ssNum) {
    this.name = name;
    this.ssNum = ssNum;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public void setSsNum(String ssNum) {
    this.ssNum = ssNum;
  }

  public String getSsNum() {
    return ssNum;
  }
}
```

This class is used by **EmployeeSearchService** for transferring employee search results data from the Model (**EmployeeSearchService**) to the View (**search.jsp**). Oftentimes, this "transfer" object is referred to as a Data Transfer Object (DTO) or Value Object (VO) and has the simple responsibility of being a data container and abstracting the Model from the View.

web.xml

The **web.xml** file, shown next, is a standard Web Archive deployment descriptor used to configure the Mini HR application. Because the file contains several configuration details, it will be reviewed section by section.

```
<?xml version="1.0"?>

<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
```

```
<web-app>

  <!-- Action Servlet Configuration -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

  <!-- The Welcome File List -->
  <welcome-file-list>
    <welcome-file>/index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

The following is the first section of the **web.xml** file. It declares the Struts Controller servlet, **ActionServlet**, and configures it.

```
<!-- Action Servlet Configuration -->
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

This declaration starts by assigning a name to the servlet that will be used in the next section for mapping the servlet to specific application requests. After defining the servlet's name and class,

the **config** initialization parameter is defined. This parameter informs the Struts **ActionServlet** where to find its central configuration file: **struts-config.xml**. Finally, the **<load-on-startup>** tag is used to specify the order in which servlets are loaded for a given Web application when the servlet container starts. The value specified with the **<load-on-startup>** tag is essentially a priority, and servlets with a higher priority (lower value) are loaded first.

The second section of the **web.xml** file causes **ActionServlet** to respond to certain URLs:

```
<!-- Action Servlet Mapping -->
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

Notice that the **<servlet-name>** tag references the same name declared in the preceding section. This associates the previous servlet declaration with this mapping. Next, the

<url-pattern> tag is used to declare the URLs that **ActionServlet** will respond to. In this case, it is saying that **ActionServlet** will process any requests for pages that end in **.do**. So, for example, a request to

http://localhost:8080/MiniHR/page.do

or a request to

http://localhost:8080/MiniHR/dir1/dir2/page2.do

will be routed to the Struts **ActionServlet** for processing.

The final section of the **web.xml** file declares the Welcome File list that the Mini HR application will use:

```
<!-- The Welcome File List -->
<welcome-file-list>
  <welcome-file>/index.jsp</welcome-file>
</welcome-file-list>
```

The Welcome File list is a list of files that the Web server will attempt to respond with when a given request to the Web application goes unfulfilled. For example, in Mini HR's case, you can enter a URL of **http://localhost:8080/MiniHR/** and **index.jsp** will be executed, because no page has been specified in the URL. The servlet container detects this and references the Welcome File list for pages that should be tried to respond to the request.

In this case, the servlet container will try to respond with a page at **/index.jsp**. If that page is unavailable, an error will be returned. Note that the Welcome File list can encompass several pages. In that case, the servlet container will iterate through the list until a file is found that can be served for the request.

struts-config.xml

The **struts-config.xml** file, shown next, is the central location for all of a Struts application's configuration settings. Recall from the previous description of the **web.xml** file that the **struts-config.xml** file is used by **ActionServlet** to configure the application. The basic configuration

information is covered here, but a complete description will have to wait until you know more about Struts. (A complete discussion of configuration is found in Chapter 18.)

```
<?xml version="1.0"?>

<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.3//EN"
  "http://struts.apache.org/dtds/struts-config_1_3.dtd">

<struts-config>

  <!-- Form Beans Configuration -->
  <form-beans>
      <form-bean name="searchForm"
                 type="com.jamesholmes.minihr.SearchForm"/>
  </form-beans>

  <!-- Global Forwards Configuration -->
  <global-forwards>
    <forward name="search" path="/search.jsp"/>
  </global-forwards>

  <!-- Action Mappings Configuration -->
  <action-mappings>
    <action path="/search"
            type="com.jamesholmes.minihr.SearchAction"
            name="searchForm"
            scope="request"
            validate="true"
            input="/search.jsp">
    </action>
  </action-mappings>

  <!-- Message Resources Configuration -->
  <message-resources
        parameter="com.jamesholmes.minihr.MessageResources"/>

</struts-config>
```

Struts configuration files are XML-based and should conform to the Struts Configuration Document Type Definition (DTD). The **struts-config.xml** file just shown begins by declaring its use of the Struts Configuration DTD:

```
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.3//EN"
  "http://struts.apache.org/dtds/struts-config_1_3.dtd">
```

Next, is the Form Beans Configuration section, which is used to specify all of the **ActionForm** objects used in your Struts application. In this case, only one Form Bean is being used:**SearchForm**. The definition of the Form Bean, shown here, allows you to associate a logical name or alias of "searchForm" with the **SearchForm** object:

```
<form-bean name="searchForm"
           type="com.jamesholmes.minihr.SearchForm"/>
```

That way, your application code (i.e., JSPs, **Action** objects, and so on) will reference "searchForm" and not "com.jamesholmes.minihr.SearchForm". This allows the class definition to change without causing the code that uses the definition to change.

The next section of the file, Global Forwards Configuration, lists the forward definitions that your application will have. Forward definitions are a mechanism for assigning a logical name to the location of a page. For example, for the Mini HR application, the name "search" is assigned to the "search.jsp" page:

```
<forward name="search" path="/search.jsp"/>
```

As in the case of Form Beans, the use of forward definitions allows application code to reference an alias and not the location of pages. Note that this section of the file is dedicated to "Global" forwards, which are made available to the entire Struts application. You can also specify action-specific forwards that are nested in an **<action>** tag in the config file:

```
<action ...>
  <forward .../>
</action>
```

The topic of action-specific forward definitions is examined in Chapter 5.

After the Global Forwards Configuration section comes the Action Mappings Configuration section of the file. This section is used to define the **Action** classes used in your Struts application. Remember from the previous section on **SearchAction.java** that **Action** classes are used to handle discrete Controller tasks. Because the **SearchAction** mapping, shown here, has many settings, each is examined in detail.

```
<action path="/search"
        type="com.jamesholmes.minihr.SearchAction"
        name="searchForm"
        scope="request"
        validate="true"
        input="/search.jsp">
</action>
```

The first part of the Action Mappings Configuration section defines the path associated with this action. This path corresponds to the URL used to access your Struts application. Recall from the "web.xml" section that your application is configured to have any URLs ending in **.do** be handled by **ActionServlet**. Setting the path to "/search" for this action essentially says that a request to "/search.do" should be handled by **SearchAction**. Struts removes the **.do** from the URL (resulting in "/search") and then looks in the Struts configuration file settings for an Action Mapping that corresponds to the URL.

The next **<action>** attribute, **type**, specifies the **Action** class that should be executed when the path specified with the **path** attribute is requested. The **name** attribute corresponds to the name of a Form Bean defined in the **struts-config.xml** file. In this case, "searchForm" corresponds to the Form Bean set up earlier. Using the **name** attribute informs Struts to populate the specified Form Bean with data from the incoming request. The **Action** object will then have access to the Form Bean to access the request data.

The next two attributes, **scope** and **validate**, are related to the Form Bean defined with the **name** attribute. The **scope** attribute sets the scope for the Form Bean associated with this action. For example, use "request" for request scope or "session" for session scope. The **validate** attribute is used to specify whether the Form Bean defined with the **name** attribute should have its**validate( )** method called after it has been populated with request data.

The final **<action>** attribute, **input**, is used to inform the **Action** object what page is being used to "input" data to (or execute) the Action; in this case, it is "search.jsp". Struts will forward to the page specified by the **input** attribute if validation fails.

The last section of the file, Message Resources Configuration, is used to define the location of the application resource bundle file (e.g., **MessageResources.properties**). Notice that the file is specified using Java's package mechanism: package.package.class (i.e., "com.jamesholmes.minihr.MessageResources"). This allows **ActionServlet** to load the properties file from the same place that classes are loaded. An extension of **.properties** is automatically appended to the resource bundle file name by Struts and thus should not be specified in the Struts configuration file.

MessageResources.properties

The **MessageResources.properties** file, shown next, is based on the Java Resource Bundle functionality for externalizing and internationalizing application strings, messages, and labels.

```
# Label Resources
label.search.name=Name
label.search.ssNum=Social Security Number

# Error Resources
error.search.criteria.missing=Search Criteria Missing
error.search.ssNum.invalid=Invalid Social Security Number
errors.header=<font color="red"><cTypeface:Bold>Validation
Error(s)</b></font><ul>
errors.footer=</ul><hr width="100%" size="1" noshade="true">
errors.prefix=<li>
errors.suffix=</li>
```

Notice that this file is simply composed of name-value pairs, where the name is a key and the value is a message corresponding to the key. Each of the name-value pairs is then used by the Struts application whenever a string, message, or label needs to be displayed. Externalizing these strings in a separate file instead of embedding them in the application allows the strings to be changed without having to recompile the application (separation of concerns). Externalizing the strings also allows the application to support internationalization so that it can be tailored to different locales. As you'll see in Chapter 10, internationalization with Struts is straightforward and easy with the use of resource bundle properties files for strings, messages, and labels.

Compiling, Packaging, and Running the Application

Now that you have examined the sample application in detail, it's time to compile, package, and run the application. First, though, download and install the Struts and Tomcat software if you have not already done so. Tomcat is a free servlet container available for download from the Internet and will be used in the examples in this book for running Struts applications. Tomcat is also the reference implementation for the JSP and servlet specifications from Sun. Of course, you don't have to use Tomcat to run the examples, but it is the only method described by this book. So, to follow along, it is strongly suggested that you use Tomcat.

Each of the following sections is dedicated to a step of the process. First, you will set up the Struts and Tomcat software. Next, you'll compile the application. Then, you'll package the application in a standard Web Archive file. Finally, you'll see how to deploy and run the application with Tomcat.

Downloading and Installing Struts and Tomcat

As mentioned, both Struts and Tomcat are freely available for download from the Internet. Following are the Web sites for each:

- **Struts http://struts.apache.org/**
- **Tomcat http://tomcat.apache.org/**

After you have downloaded the Struts and Tomcat software distributions, you will need to choose a directory to install them to. After selecting a directory, extract the files of each distribution to that directory. For example, if you choose to install the distributions in a directory called **c:\java**, then the Struts files would be located at **c:\java\struts-1.3.5** (or similar) and Tomcat would be installed at **c:\java\apache-tomcat-5.5.17** (or similar).

Compiling the Application

The Mini HR application consists of several files; however, only the Java source code files need to be compiled before you package and run the application. Because the Java source code files use the servlet and Struts APIs, you need to add these libraries to your Java classpath. You could do this by updating your CLASSPATH environment variable. Alternatively, you can just specify the path when you compile the Mini HR application.

In addition to the files that you created and reviewed earlier in this chapter, you also need to copy the following files to the **c:\java\MiniHR\WEB-INF\lib** directory. These **.jar** files contain the Struts and associated library class files that are necessary for the Mini HR application to run once it is packaged as a **.war** file.

```
c:\java\struts-1.3.5\lib\antlr-2.7.2.jar
c:\java\struts-1.3.5\lib\bsf-2.3.0.jar
c:\java\struts-1.3.5\lib\commons-beanutils-1.7.0.jar
```

```
c:\java\struts-1.3.5\lib\commons-chain-1.1.jar
c:\java\struts-1.3.5\lib\commons-collections-2.1.jar
c:\java\struts-1.3.5\lib\commons-digester-1.6.jar
c:\java\struts-1.3.5\lib\commons-fileupload-1.1.1.jar
c:\java\struts-1.3.5\lib\commons-io-1.1.jar
c:\java\struts-1.3.5\lib\commons-logging-1.0.4.jar
c:\java\struts-1.3.5\lib\commons-validator-1.3.0.jar
c:\java\struts-1.3.5\lib\oro-2.0.8.jar
c:\java\struts-1.3.5\lib\struts-core-1.3.5.jar
c:\java\struts-1.3.5\lib\struts-el-1.3.5.jar
c:\java\struts-1.3.5\lib\struts-extras-1.3.5.jar
c:\java\struts-1.3.5\lib\struts-faces-1.3.5.jar
c:\java\struts-1.3.5\lib\struts-scripting-1.3.5.jar
c:\java\struts-1.3.5\lib\struts-taglib-1.3.5.jar
c:\java\struts-1.3.5\lib\struts-tiles-1.3.5.jar
```

Note The preceding **.jar** files are those packaged with Struts 1.3.5 and may change over time with newer versions of Struts. If you are using a different version of Struts, you should use the**.jar** files included with that version of Struts.

Assuming that you have installed Struts at **c:\java\struts-1.3.5**, installed Tomcat at **c:\ java\apache-tomcat-5.5.17**, and placed the Mini HR application files at **c:\java\MiniHR**, the following command line will compile the Mini HR application when run from the **c:\ java\MiniHR** directory:

```
javac -classpath WEB-INF\lib\antlr-2.7.2.jar;
                 WEB-INF\lib\bsf-2.3.0.jar;
                 WEB-INF\lib\commons-beanutils-1.7.0.jar;
                 WEB-INF\lib\commons-chain-1.1.jar;
                 WEB-INF\lib\commons-collections-2.1.jar;
                 WEB-INF\lib\commons-digester-1.6.jar;
                 WEB-INF\lib\commons-fileupload-1.1.1.jar;
                 WEB-INF\lib\commons-io-1.1.jar;
                 WEB-INF\lib\commons-logging-1.0.4.jar;
                 WEB-INF\lib\commons-validator-1.3.0.jar;
                 WEB-INF\lib\oro-2.0.8.jar;
                 WEB-INF\lib\struts-core-1.3.5.jar;
                 WEB-INF\lib\struts-el-1.3.5.jar;
                 WEB-INF\lib\struts-extras-1.3.5.jar;
                 WEB-INF\lib\struts-faces-1.3.5.jar;
                 WEB-INF\lib\struts-scripting-1.3.5.jar;
                 WEB-INF\lib\struts-taglib-1.3.5.jar;
                 WEB-INF\lib\struts-tiles-1.3.5.jar;
                 C:\java\apache-tomcat-5.5.17\common\lib\servlet-api.jar
```

```
WEB-INF\src\com\jamesholmes\minihr\*.java
-d WEB-INF\classes
```

Notice that you must specify the path to each **.jar** file explicitly. Of course, if you update CLASSPATH, this explicit specification is not needed. You should also notice that the compiled code will be placed into the **WEB-INF\classes** directory, as specified by the **-d WEB-INF\classes** section of the command line. Remember from the earlier discussion that this is the standard Web Archive directory that servlet containers will look in for compiled Web application code.

To simplify the process of building the Mini HR application in this chapter and in subsequent exercises in the book, on Windows-based systems you can place the preceding command line into a batch file named **build.bat**. The **build.bat** batch file can then be used to compile the application. All you have to do is type "build" from the command line and the **build.bat** batch file will be run. For Unix/Linux-based systems you can place the command line in a simple shell script to achieve similar build "automation."

Using Ant to Compile the Application

Manually compiling Java applications from the command line can be error prone and tedious. A popular alternative solution for compiling Java code is to use Apache Ant (http://ant.apache.org/). Ant is a Java-based build tool driven by an XML-based build file. The build file includes specific targets that correspond to steps in the build process, such as compiling classes, creating the distribution (such as a **.jar** file or **.war** file), and deploying the application. A full explanation of Ant is outside the scope of this book; however, a sample ant build file, **build.xml**, is given next for compiling the Mini HR application.

```
<project name="MiniHR" default="compile" basedir=".">
  <property name="src.dir" location="src"/>
  <property name="classes.dir" location="classes"/>
  <property name="struts.lib.dir"
        location="c:/java/struts-1.3.5/lib"/>
  <property name="tomcat.lib.dir"
        location="c:/java/apache-tomcat-5.5.17/common/lib"/>

  <target name="compile">
    <javac srcdir="${src.dir}" destdir="${classes.dir}" debug="on">
      <classpath>
        <pathelement path="${classpath}"/>
        <pathelement location="${tomcat.lib.dir}/servlet-api.jar"/>
        <fileset dir="${struts.lib.dir}">
          <include name="**/*.jar"/>
        </fileset>
      </classpath>
    </javac>
  </target>
</project>
```

To run the Ant build file, simply type "ant" from the directory where your **build.xml** file is located. By default, Ant looks for a file named **build.xml** and runs it.

Packaging the Application

Because Struts applications are standard Java EE Web applications, this application will be packaged using the standard Web Archive format. Packaging the application as a **.war** file allows the application to be easily deployed on any Java EE–compliant servlet container with ease. Because you arranged the files for the Mini HR application in the standard Web Archive directory structure, packaging them into a **.war** file is straightforward.

Following is the command line for creating a **MiniHR.war** file, assuming that you run the command from the Mini HR application directory (c:\java\MiniHR):

```
jar cf MiniHR.war *
```

After you run the command, a **MiniHR.war** file will be created and ready for deployment.

Running the Application

Once you have packaged your application, running it is as simple as placing the Web Archive file into Tomcat's **webapps** directory and then starting up Tomcat. By default, Tomcat starts up on port 8080, and thus the server can be accessed at **http://localhost:8080/**. To access the Mini HR application, point your browser to **http://localhost:8080/MiniHR/**. You'll notice that the name of your Web Archive file is used for the URL of your application. Because you packaged the Mini HR application in a file called **MiniHR.war**, **/MiniHR/** is used for the application's URL.

When you first access the **http://localhost:8080/MiniHR/** URL, **index.jsp** will be run, because it was specified as the Welcome File in the **web.xml** deployment descriptor. From the opening page, select the Search for Employees link. The Search page allows you to search for employees by name or social security number. If you do not enter any search criteria, an error message will be shown on the page. Similarly, if you enter an invalid social security number, an error message will be shown on the page after you click the Search button.

Figures 2-2 through 2-5 show Mini HR in action.

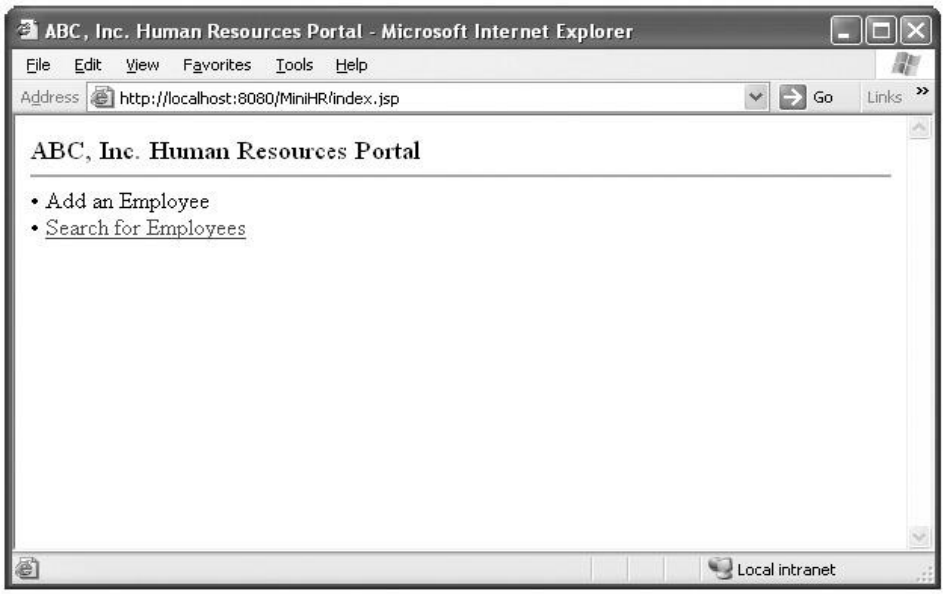


Figure 2-2: The opening screen

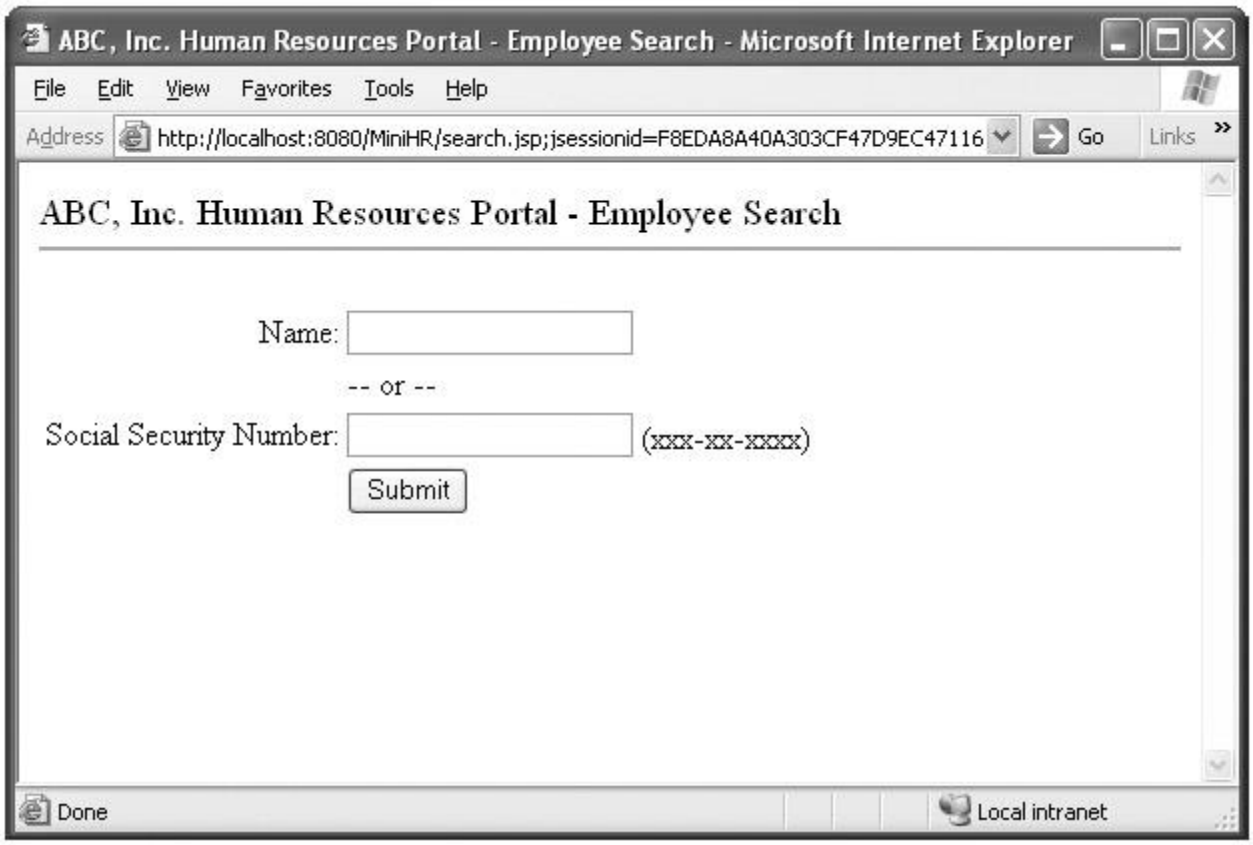


Figure 2-3: The Employee Search screen

Figure 2-4: The Employee Search screen with a validation error

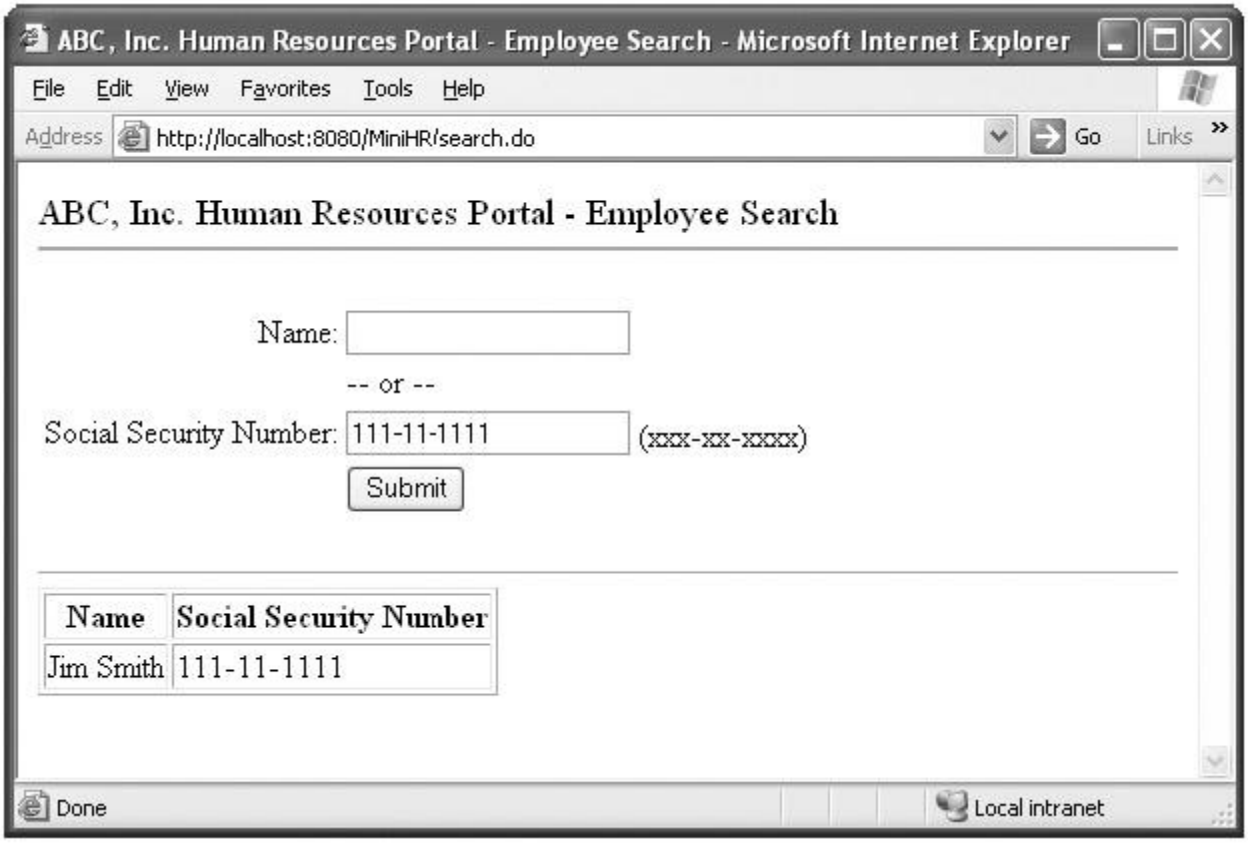


Figure 2-5: The Employee Search screen with search results

Understanding the Flow of Execution

Before leaving the Mini HR example, it is necessary to describe the way that execution takes place. As explained in Chapter 1, Struts uses the Model-View-Controller design pattern. The MVC architecture defines a specific flow of execution. An understanding of this flow of execution is crucial to an overall understanding of Struts.

For Mini HR, execution proceeds in the following way:

1. The browser makes a request to Mini HR for an employee lookup. This request is processed by **ActionServlet** (Controller).
2. **ActionServlet** populates the **SearchForm** object (View) with the search criteria entered on the **search.jsp** page.
3. **ActionServlet** delegates request-specific processing to the **SearchAction** object (Controller).
4. **SearchAction** (Controller) interfaces with the **EmployeeSearchService** object (Model) to perform the employee search. **EmployeeSearchService** returns an **ArrayList** of**Employee** objects (Model).
5. **SearchAction** (Controller) forwards control to **search.jsp** (View).
6. **search.jsp** (View) uses the **ArrayList** of **Employee** objects (Model) to generate a response to the browser.

The flow of execution for Mini HR can be generalized for any Struts application as shown here. Figure 2-6 shows the flow execution in graphic form.

1. The browser makes a request to the Struts application that is processed by **ActionServlet** (Controller).
2. **ActionServlet** (Controller) populates the **ActionForm** (View) object with HTML form data and invokes its **validate( )** method.
3. **ActionServlet** (Controller) executes the **Action** object (Controller).

4. **Action** (Controller) interfaces with model components and prepares data for view.
5. **Action** (Controller) forwards control to the JSP (View).
6. JSP (View) uses model data to generate a response to the browser.

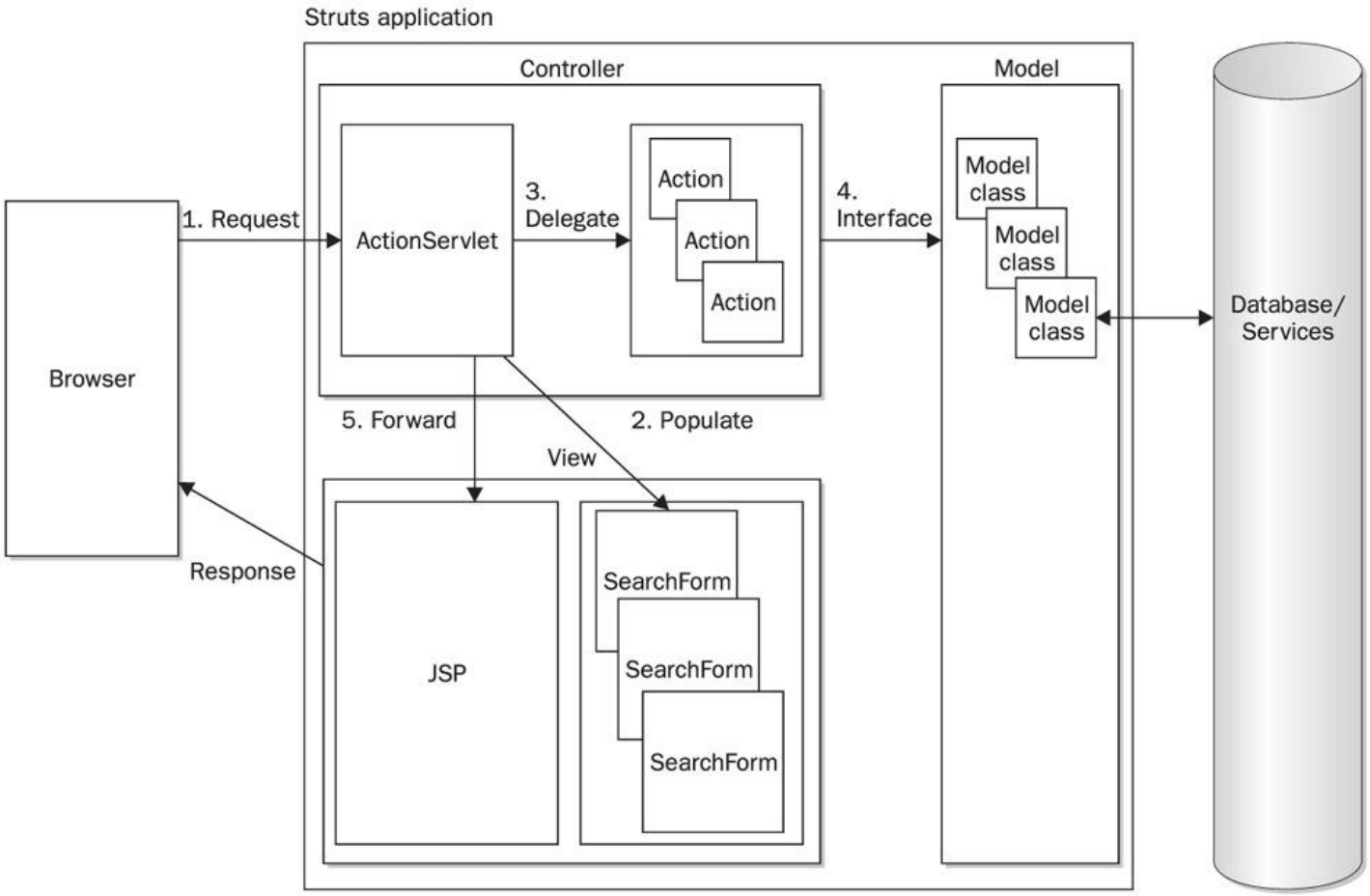


Figure 2-6: Flow of execution

Remember, the same basic pattern of execution applies to any Struts application.

Now that you understand the basic structure of a Struts application and how its components work together, it's time to move on to an in-depth examination of Struts. As mentioned at the start, all of the topics presented in this chapter are examined in detail in the chapters that follow.

Chapter 3: The Model Layer

As you know, Struts is a framework that is used to build applications based on the Model-View-Controller (MVC) architecture. Because the MVC organization is at the foundation of Struts, it is not possible to fully utilize Struts without a clear understanding of each part of the MVC architecture. Therefore, this and the following two chapters examine in depth the Model, View, and Controller portions of a Struts application, beginning in this chapter with the Model.

What Is the Model?

In an MVC application, the Model layer is typically the largest and most important piece. The Model is designed to house the business logic and data access code; in other words, the Model consists of the core of the application. For example, if an application computes the average sell-through rate of a

product, the Model layer performs that computation. For an application that maintains a database of employees, complete with salary and tax information, the Model handles the maintenance task. Thus, it is the Model that defines what the application does. The View and Controller interact with the Model and provide a user interface to it.

The MVC architecture dictates that the Model layer should be self-contained and function independently from the View and Controller layers. That way, the core application code can be used over and over again with multiple user interfaces. For example, you could have a Web interface for the application as well as a stand-alone or wireless interface. Each interface (Web, stand-alone, and so on) would have its own code for the user interface (View) but would reuse the core application (Model) code. This is the basis for the MVC architecture: having a clean separation of responsibilities and reducing coupling between application layers.

Model Layer Breakdown

The typical Model layer of a correctly designed MVC application can be broken down into three conceptual sublayers. Each sublayer can be thought of as a component or responsibility of the Model. Figure 3-1 illustrates this breakdown.

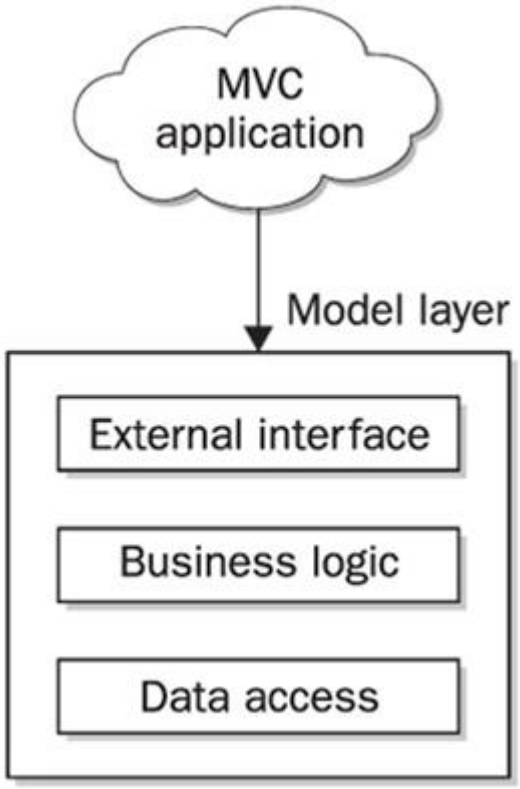


Figure 3-1: Model layer breakdown

Each sublayer does not necessarily represent a separate set of classes, but rather the Model's set of responsibilities. You may choose to house a specific function's code for all layers in one large class, or you may break down each sublayer into fine-grained objects. The level of object granularity is up to you, and what's best and/or necessary really depends on the size and complexity of your application. The following are the three sublayers:

- **External interface** Composed of code that provides an interface that external code uses to interact with the Model.
- **Business logic** Encompasses the bulk of the Model code and provides the business functionality for an application.

- **Data access** Composed of code for communicating with an application's data sources such as a database.

Struts and the Model

The Struts framework does not provide any specific features or constraints for developing the Model layer of your application. At first glance this may seem odd, or even a shortcoming, given that Struts is designed for building MVC applications. However, it's actually a key design feature of Struts and is a great benefit. By not dictating how the Model layer should be built, Struts gives your application the flexibility to use any approach or technology for building the Model layer code. Whether it be an Object-Relational Mapping (ORM) framework (e.g., Hibernate or TopLink), Enterprise JavaBeans (EJB), Java Data Objects (JDO), or the Data Access Objects (DAO) pattern, Struts will accommodate.

Because the Model defines the business logic, the Model is where Struts ends and your application code begins. Your Model code will be accessed from subclasses of the Struts **Action**object that are part of the Controller layer of the Struts framework. **Action** subclasses interact with the Model via interfaces and use its Data Transfer Objects to pass and retrieve data.

You should not place any business logic or data access code in **Action** objects. Doing so would bypass the separation of the Model and the Controller. Similarly, your Model code should not have any references to Struts code or objects. Violating this rule unnecessarily couples your core application code to Struts.

Using BeanUtils to Transfer Data to Model Classes

As stated, Model code should not have any references to Struts code to prevent from coupling your application code to Struts. This guideline can create a lot of headache for developers when accessing Model layer code. Form Beans are used to collect form data which eventually must be transferred to the Model layer code for processing. Because it is bad practice to pass a Form Bean directly to the Model layer code, you must transfer the data from the Form Bean to a Model Data Transfer Object to be transferred to the Model layer code. As you can imagine, it is cumbersome to write several "copy" statements that simply copy data from one object to the other, as shown in the following example:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public final class ExampleAction extends Action
{
  public ActionForward execute(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
```

```
    throws Exception
  {
    ExampleForm exampleForm = (ExampleForm) form;

    Example example = new Example();
    example.setField1(exampleForm.getField1());
    example.setField2(exampleForm.getField2());
    example.setField3(exampleForm.getField3());
    example.setField4(exampleForm.getField4());
    example.setField5(exampleForm.getField5());

    ExampleService service = new ExampleService();
    service.updateExample(example);

    return mapping.findForward("success");
  }
}
```

Instead of writing tedious, repetitive "copy" statements, you can use the Jakarta Commons BeanUtils library to copy all of the properties for you, as shown next.

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.commmons.beanutils.PropertyUtils;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public final class ExampleAction extends Action
{
  public ActionForward execute(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    ExampleForm exampleForm = (ExampleForm) form;
```

```
    Example example = new Example();
    PropertyUtils.copyProperties(example, exampleForm);

    ExampleService service = new ExampleService();
    service.updateExample(example);

    return mapping.findForward("success");
  }
}
```

Struts comes packaged with the Jakarta Commons BeanUtils library and makes use of it heavily in the core framework code, so you do not have to download BeanUtils separately. All you have to do is import the **org.apache.commons.beanutils.PropertyUtils** class and use its **copyProperties( )** method to transfer fields from one object to the other. The destination object is the first parameter to the **copyProperties( )** method, and the origination object is the second method parameter.
The **copyProperties( )** method will automatically copy all fields from the origination object to the destination object that have the same name. For example, if the origination object has two fields named "firstName" and "lastName" respectively, the **copyProperties( )** method will attempt to find the same two fields on the destination object and call their corresponding setter methods.

Note For more information on the Jakarta Commons BeanUtils project, visit the BeanUtils Web site at: http://jakarta.apache.org/commons/beanutils/. Alternatively, the FormDef extension for Struts allows you to use Model objects directly instead of creating Form Beans and transferring data between the objects. Appendix B has a brief introduction to FormDef.

Reviewing the Model Layer of the Mini HR Application

Because Struts has little to do with the Model layer, there is little more to say about it. However, before moving on, it will be helpful to review the Model layer of the Mini HR application developed in Chapter 2. Doing so clearly illustrates how the Model code is separate from the rest of the application.

Mini HR's Model layer consists of two classes: **EmployeeSearchService** and **Employee**.
The **EmployeeSearchService** class is shown next:

```
package com.jamesholmes.minihr;

import java.util.ArrayList;

public class EmployeeSearchService
{
  /* Hard-coded sample data. Normally this would come from a real data
     source such as a database. */
  private static Employee[] employees =
  {
    new Employee("Bob Davidson", "123-45-6789"),
    new Employee("Mary Williams", "987-65-4321"),
```

```
    new Employee("Jim Smith", "111-11-1111"),
    new Employee("Beverly Harris", "222-22-2222"),
    new Employee("Thomas Frank", "333-33-3333"),
    new Employee("Jim Davidson", "444-44-4444")
  };

  // Search for employees by name.
  public ArrayList searchByName(String name) {
    ArrayList resultList = new ArrayList();

    for (int i = 0; i < employees.length; i++) {
      if(employees[i].getName().toUpperCase().indexOf(name.toUpperCase())
         != -1)
      {
        resultList.add(employees[i]);
      }
    }

     return resultList;
  }

  // Search for employee by social security number.
  public ArrayList searchBySsNum(String ssNum) {
    ArrayList resultList = new ArrayList();

    for (int i = 0; i < employees.length; i++) {
      if (employees[i].getSsNum().equals(ssNum)) {
        resultList.add(employees[i]);
      }
    }

    return resultList;
  }
}
```

EmployeeSearchService fulfills all three of the model's sublayers: external interface, business logic, and data access. The external interface is defined by the methods **searchByName( )** and**searchBySsNum( )**. The business logic is contained in the implementation to those methods, which finds an employee based on either his or her name or social security number. Data access occurs each time the hard-coded **Employee** array is used.

In a small, sample application such as Mini HR, there is nothing wrong with implementing the entire Model within **EmployeeSearchService**. However, in a more complicated application, a class such as this would normally be used as only the external interface to the Model. In this approach, it would house only skeletal **searchByName( )** and **searchBySsNum( )** methods, which would pass through (or *delegate*) requests to the business logic sublayer where their actual implementation would exist. For example, the business logic sublayer code could be implemented in a class named **EmployeeSearchImpl**. Furthermore, **EmployeeSearchImpl** would then communicate with data access sublayer classes to actually query employee data from a database or such, rather than containing the data access code itself.

The **Employee** class, shown next, is a *domain* object used to represent an entity in the model. In this scenario it is used as a conduit for transferring data to and from the Model. As such, it can be thought of as being part of the Model layer.

```
package com.jamesholmes.minihr;


public class Employee
{
  private String name;
  private String ssNum;

  public Employee(String name, String ssNum) {
    this.name = name;
    this.ssNum = ssNum;
  }

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }

  public void setSsNum(String ssNum) {
    this.ssNum = ssNum;
  }

  public String getSsNum() {
    return ssNum;
  }
}
```

Typically, an object of this type is referred to as a Data Transfer Object (DTO) or Value Object (VO) and is part of the external interface sublayer. The Model uses DTOs to send data back through its external interface and to accept data through its external interface. You can think of these classes as interfaces themselves because they specify the format and packaging of the data expected by the Model.

Chapter 4: The View Layer

Overview

In an MVC application, the View layer provides an interface to your application, be it for users with a browser or for another application using something like Web services. Basically the View layer is the conduit for getting data in and out of the application. It does not contain business logic, such as calculating interest for a banking application or storing items in a shopping cart for an online catalog. The View layer also does not contain any code for persisting data to or retrieving data from a data source. Rather, it is the Model layer that manages business logic and data access. The View layer simply concentrates on the interface.

Keeping the Model and View layers separate from one another allows an application's interface to change independent of the Model layer and vice versa. This separation also allows the application to have multiple interfaces (or views). For instance, an application could have a Web interface and a wireless interface. In this case, each interface is separate, but both use the same Model layer code without the Model layer being tied to either interface or either interface having to know about the other interface.

Struts and the View Layer

Struts provides a rich set of functionality and features for developing the View layer of MVC applications. There are several forms that the View layer of a Struts application can take. It can be HTML/JSP (the most common case) or it can be XML/XSLT, Velocity, Swing, or whatever your application requires. This is the power of Struts and MVC. Because HTML/JSP is the typical View technology used for Java-based Web applications, Struts provides the most functionality and features for developing your application this way. The remainder of this chapter focuses on Struts' support for creating the View layer using HTML/JSP.

Struts' HTML/JSP View layer support can be broken down into the following major components:

- JSP pages
- Form Beans
- JSP tag libraries
- Resource bundles

Each of these components is examined in detail in this chapter, but first it is helpful to understand how they fit together in the View layer.

JSP pages are at the center of the View components; they contain the HTML that is sent to browsers for users to see and they contain JSP library tags. The library tags are used to retrieve data from Form Beans and to generate HTML forms that, when submitted, will populate Form Beans. Additionally, library tags are used to retrieve content from resource bundles. Together, all of the Struts View layer components are used to generate HTML that browsers render. This is what the user sees.

On the back side, the View layer populates Form Beans with data coming from the HTML interface. The Controller layer then takes the Form Beans and manages getting their data and putting it into the Model layer. Additionally, the Controller layer takes data from the Model layer and populates Form Beans so that the data can be presented in the View layer.

The following sections explain each of these major View components in detail.

JSP Pages

JSPs are the centerpiece of the Struts View layer. They contain the static HTML and JSP library tags that generate dynamic HTML. Together the static and dynamically generated HTML gets sent to the user's browser for rendering. That is, the JSPs contain the code for the user interface with which a user interacts.

JSPs in Struts applications are like JSPs in any other Java-based Web application. However, to adhere to the MVC paradigm, the JSPs should not contain any code for performing business logic or code for directly accessing data sources. Instead, the JSPs are intended to be used solely for displaying data and capturing data. Struts provides a set of tag libraries that supports displaying data and creating HTML forms that capture data. Additionally, the tags support displaying content stored in resource bundles. Therefore, JSPs (coupled with Form Beans) provide the bulk of the Struts View layer. The JSP tag libraries glue those two together and the resource bundles provide a means of content management.

Form Beans

Form Beans provide the conduit for transferring data between the View and Controller layers of Struts applications. When HTML forms are submitted to a Struts application, Struts takes the incoming form data and uses it to populate the form's corresponding Form Bean. The Struts Controller layer then uses the Form Beans to access data that must be sent to the Model layer. On the flip side, the Controller layer populates Form Beans with Model layer data so that it can be displayed with the View layer. Essentially, Form Beans are simple data containers. They either contain data from an HTML form that is headed to the Model via the Controller or contain data from the Model headed to the View via the Controller.

Form Beans are basic Java beans with getter and setter methods for each of their properties, allowing their data to be set and retrieved easily. The **org.apache.struts.action.ActionForm** class is the base abstract class that all Form Beans must descend from (be it directly or via a subclass). Because Form Beans are simple data containers, they are principally composed of fields, and getter and setter methods for those fields. Business logic and data access code should not be placed in these classes. That code goes in Model layer classes. The only other methods that should be in these classes are helper methods or methods that override **ActionForm**'s base **reset( )** and **validate( )** methods.

Note Form Beans are based on the JavaBeans specification and must have proper getter and setter methods for each field in order for introspection by the Struts framework to function properly. For more information on the JavaBeans specification visit: http://java.sun.com/products/javabeans/.

The **ActionForm** class has a **reset( )** method and a **validate( )** method that are intended to be overridden by subclasses where necessary. The **reset( )** method is a hook that Struts calls before the Form Bean is populated from an application request (e.g., HTML form submission). The **validate( )** method is a hook that Struts calls after the Form Bean has been populated from an application request. Both of these methods are described in detail later in this section.

Following is an example Form Bean:

```
import org.apache.struts.action.ActionForm;

public class EmployeeForm extends ActionForm
{
  private String firstName;
  private String lastName;
  private String department;

  public void setFirstName(String firstName) {
    this.firstName = firstName;
  }

  public String getFirstName() {
    return firstName;
  }

  public void setLastName(String lastName) {
    this.lastName = lastName;
  }

  public String getLastName() {
    return lastName;
  }

  public void setDepartment(String department) {
    this.department = department;
  }

  public String getDepartment() {
    return department;
  }
}
```

Form Bean properties can be of any object type, be it a built-in class like **String**, **Integer**, or **Boolean** or a complex application-specific class such as an **Address** object that has fields for street address, city, state, and ZIP. Struts uses reflection to populate the Form Beans and can traverse nested object hierarchies to any level so long as the getter and setter methods are public. For example, if your Form Bean had an **Address** object field named **address**, to access the **city** field on the **Address** object the Form Bean would need a public **getAddress( )** method that returned an **Address** object. The **Address** object would need a public **getCity( )** method that would return

a **String**. Properties that are themselves objects with properties are known as*nested properties.* Nested properties is the name given to properties that represent an object hierarchy.

Often, it's best to have Form Bean fields be **String**s instead of other types. For example, instead of having an **Integer**-type field for storing a number, it's best to use a **String**-type field. This is because all HTML form data comes in the form of strings. If a letter rather than a number is entered in a numeric field, it's better to store the value in a **String** so that the original data can be returned to the form for correcting. If instead the data is stored in a **Long**, when Struts attempts to convert the string value to a number, it will throw a **NumberFormatException** if the value is a letter. Then, when the form is redisplayed showing the invalid data, it will show 0 instead of the originally entered value, because letters cannot be stored in numeric-type fields.

Note Although Struts best practices dictate that all Form Bean fields should be of type **String**, the simplicity of using types that more naturally match the data sometimes prevails. For this scenario, the Struts **ActionServlet** has an initialization parameter, **convertNull**, that informs Struts to default to null for Java wrapper type classes (e.g., null instead of 0 for numeric types).

Configuring Form Beans

To use Form Beans, you have to configure them in the Struts configuration file. Following is a basic Form Bean definition:

```
<!-- Form Beans Configuration -->
<form-beans>
  <form-bean name="searchForm"
             type="com.jamesholmes.minihr.SearchForm"/>
</form-beans>
```

Form Bean definitions specify a logical name and the class type for a Form Bean. Once defined, Form Beans are associated with actions by action mapping definitions, as shown next:

```
<!-- Action Mappings Configuration -->
<action-mappings>
  <action path="/search"
          type="com.jamesholmes.minihr.SearchAction"
          name="searchForm"
          scope="request"
          validate="true"
          input="/search.jsp">
  </action>
</action-mappings>
```

Actions specify their associated Form Bean with the **name** attribute of the **action** tag, as shown in the preceding snippet. The value specified for the **name** attribute is the logical name of a Form Bean defined with the **form-bean** tag. The **action** tag also has a **scope** attribute to specify the scope that the Form Bean will be stored in and a **validate** attribute to specify whether the Form Bean's **validate( )** method should be invoked after the Form Bean is populated. The **input** attribute of the **action** tag is typically used to specify a path that Struts should forward to if the**validate( )** method generates any errors.

The reset( ) Method

As previously stated, the abstract **ActionForm** class has a **reset( )** method that subclasses can override. The **reset( )** method is a hook that gets called before a Form Bean is populated with request data from an HTML form. This method hook was designed to account for a shortcoming in the way the HTML specification dictates that browsers should handle check boxes. Browsers send the value of a check box only if it is checked when the HTML form is submitted. For example, consider an HTML form with a check box for whether or not a file is read-only:

```
<input type="checkbox" name="readonly" value="true">
```

When the form containing this check box is submitted, the value of "true" is sent to the server only if the check box is checked. If the check box is not checked, no value is sent.

For most cases, this behavior is fine; however, it is problematic when Form Bean **boolean** properties have a default value of "true." For example, consider the read-only file scenario again. If your application has a Form Bean with a read-only property set to true and the Form Bean is used to populate a form with default settings, the read-only property will set the read-only check box's state to checked when it is rendered. If a user decides to uncheck the check box and then submits the form, no value will be sent to the server to indicate that the check box has been unchecked (i.e., set to false). By using the **reset( )** method, this can be solved by setting all properties tied to check boxes to false before the Form Bean is populated. Following is an example implementation of a Form Bean with a **reset( )** method that accounts for unchecked check boxes:

```
import org.apache.struts.action.ActionForm;


public class FileForm extends ActionForm
{
  private boolean readOnly;

  public void setReadOnly(boolean readOnly) {
    this.readOnly = readOnly;
  }

  public boolean getReadOnly() {
    return readOnly;
  }

  public void reset() {
    readOnly = false;
  }
}
```

The **reset( )** method in this example class ensures that the **readOnly** property is set to false before the form is populated. Having the **reset( )** method hook is equivalent to having the HTML form actually send a value for unchecked check boxes.

A side benefit of the **reset( )** method hook is that it offers a convenient place to reset data between requests when using Form Beans that are stored in session scope. When Form Beans are stored in session scope, they persist across multiple requests. This solution is most often used for wizard-style

process flows. Sometimes it's necessary to reset data between requests, and the **reset( )** method provides a convenient place for doing this.

The validate( ) Method

In addition to the **reset( )** method hook, the **ActionForm** class provides a **validate( )** method hook that can be overridden by subclasses to perform validations on incoming form data. The**validate( )** method hook gets called after a Form Bean has been populated with incoming form data. Following is the method signature for the **validate( )** method:

```
public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request)
```

Notice that the **validate( )** method has a return type of **ActionErrors**. The **org.apache.struts.action.ActionErrors** class is a Struts class that is used for storing validation errors that have occurred in the **validate( )** method. If all validations in the **validate( )** method pass, a return value of null indicates to Struts that no errors occurred.

Note Data validations can be performed in the **execute( )** method of action classes; however, having them in Form Beans allows them to be reused across multiple actions where more than one action uses the same Form Bean. Having the validation code in each action would be redundant.

Following is an example Form Bean with a **validate( )** method:

```
import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;

public class NameForm extends ActionForm
{
  private String name;

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
     return name;
  }

  public ActionErrors validate(ActionMapping mapping,
    HttpServletRequest request)
  {
    if (name == null || name.length() < 1) {
```

```
      ActionErrors errors = new ActionErrors();
      errors.add("name",
        new ActionMessage("error.name.required"));
      return errors;
    }

    return null;
  }
}
```

This example Form Bean has one field, **name**, that is validated in the **validate( )** method. The **validate( )** method checks whether or not the **name** field is empty. If it is empty, it returns an error indicating that fact. The **ActionErrors** object is basically a collection class for storing **org.apache.struts.action.ActionMessage** instances. Each validation inside the **validate( )** method creates an **ActionMessage** instance that gets stored in the **ActionErrors** object. The **ActionMessage** class takes a key to an error message stored in the application resource bundle. Struts uses the key to look up the corresponding error message. Furthermore, the **ActionMessage** class has constructors that take additional arguments that contain replacement values for the error message associated with the specified key.

Struts also has a built-in Validator framework that greatly simplifies performing data validations. The Validator framework allows you to declaratively configure in an XML file the validations that should be applied to Form Beans. For more information on the Validator framework, see Chapter 6.

Note The **validate( )** method will be called unless the Action Mapping has been configured with "validate=false" in the Struts configuration file.

The Lifecycle of Form Beans

Form Beans have a defined lifecycle in Struts applications. To fully understand how Form Beans work, it's necessary to understand this lifecycle. The Form Bean lifecycle is shown in Figure 4-1.

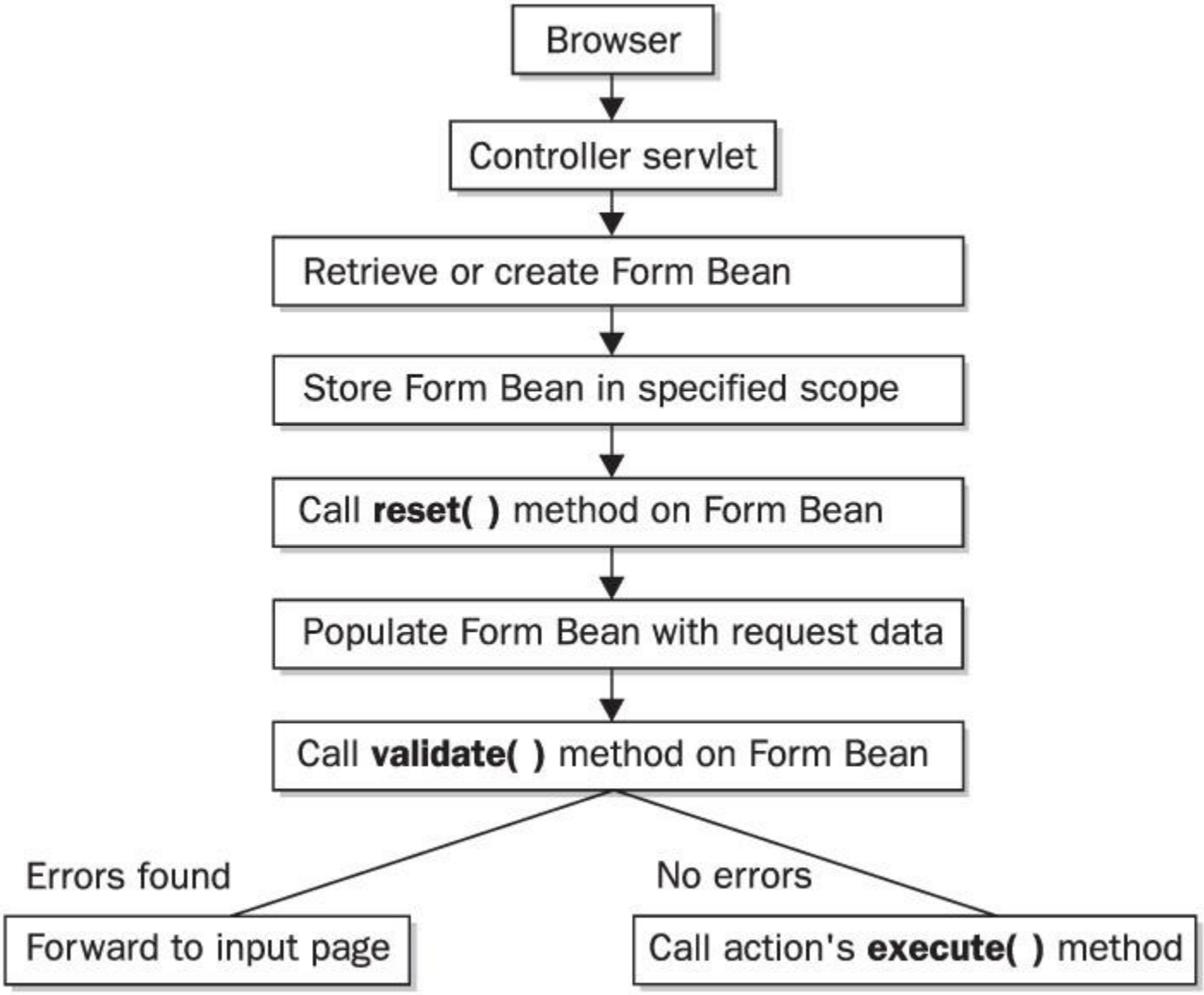


Figure 4-1: The Form Bean lifecycle

Following is an explanation of the Form Bean lifecycle. When a request is received by the Struts controller servlet, Struts maps the request to an action class that is delegated to process the request. If the action being delegated to has an associated Form Bean, Struts attempts to look up the specified Form Bean in request or session scope, based on how the action is configured in the Struts configuration file. If an instance of the Form Bean is not found in the specified scope, an instance is created and placed in the specified scope. Next, Struts calls the **reset( )**method on the Form Bean so that any processing is executed that needs to occur before the Form Bean is populated. After that, Struts populates the Form Bean with data from the incoming request. Next, the Form Bean's **validate( )** method is called. The next step in the process is based on the return value from the **validate( )** method. If the **validate( )** method records any errors and subsequently returns a non-null **ActionErrors** object, Struts forwards back to the action's input page. If, however, the return value from the **validate( )** method is null, Struts continues processing the request by calling the action's **execute( )** method.

Dynamic Form Beans

A useful addition to the 1.1 release of Struts was the introduction of Dynamic Form Beans. Dynamic Form Beans are an extension of Form Beans that allows you to specify their properties inside the

Struts configuration file instead of having to create a concrete class, with a getter and setter method for each property. The concept of Dynamic Form Beans originated because many developers found it tedious to create for every page a Form Bean that had a getter method and a setter method for each of the fields on the page's HTML form. Using Dynamic Form Beans allows the properties to be specified in a Struts configuration file. To change a property, simply update the configuration file. No code has to be recompiled.

The following snippet illustrates how Dynamic Form Beans are configured in the Struts configuration file:

```
<!-- Form Beans Configuration -->
<form-beans>
  <form-bean name="employeeForm"
             type="org.apache.struts.action.DynaActionForm">
    <form-property name="firstName"
                   type="java.lang.String"/>
    <form-property name="lastName"
                   type="java.lang.String"/>
    <form-property name="department"
                   type="java.lang.String"/>
  </form-bean>
</form-beans>
```

Dynamic Form Beans are declared in the same way as standard Form Beans, by using the **form-bean** tag. The difference is that the type of the Form Bean specified with the **form-bean** tag's**type** attribute must be **org.apache.struts.action.DynaActionForm** or a subclass thereof. Additionally, the properties for Dynamic Form Beans are specified by nesting **form-property** tags beneath the **form-bean** tag. Each property specifies its name and class type. Furthermore, an initial value for the property can be specified using the **form-property** tag's **initial** attribute, as shown next:

```
<form-property name="department"
               type="java.lang.String"
               initial="Engineering"/>
```

If an initial value is not supplied for a property, Struts sets the initial value using Java's initialization conventions. That is, numbers are set to zero, objects are set to null, and so on.

Because you declare Dynamic Form Beans in the Struts configuration file instead of creating concrete classes that extend **ActionForm**, you do not define **reset( )** or **validate( )** methods for the Dynamic Form Beans. The **reset( )** method is no longer necessary for setting default values because the **initial** attribute on the **form-property** tag achieves the same effect. The**DynaActionForm** class's implementation of the **reset( )** method resets all properties to their initial value when it is called. You can either code the functionality of the **validate( )** method inside action classes or use the Validator framework for validation. These two options eliminate the need to create a **validate( )** method on the Form Bean. If, however, you have a special case where you need to have an implementation of the **reset( )** and/or **validate( )** method for your Dynamic Form Bean, you can subclass **DynaActionForm** and create the methods there. Simply specify your **DynaActionForm** subclass as the type of the Form Bean in the Struts configuration file to use it.

While Dynamic Form Beans shift the declaration of Form Beans and their fields from concrete classes to the Struts configuration file, an argument can be made that they don't save that much time. You still have to explicitly define each field for the Form Beans in the Struts configuration file. Thus you are really only getting the benefit of not having to recompile classes each time a change is made to the definition of a Form Bean. To further reduce the amount of overheard required in creating Form Beans, you can use what's known as a *Lazy DynaBean.* Lazy DynaBeans come to Struts by way of the Jakarta Commons BeanUtils project that Struts uses throughout the framework for bean manipulation. As

of Struts 1.2.4, you can simply declare a name for a Form Bean in the Struts configuration file and specify its type as **org.apache.commons.beanutils.LazyDynaBean** and no other configuration is needed. The Lazy DynaBean will accommodate any form properties sent to it from an HTML Form. That is, no fields need to be explicitly declared for the Form Bean. An example of configuring a Lazy DynaBean is shown here:

```
<form-beans>
  <form-bean name="employeeForm"
             type="org.apache.commons.beanutils.LazyDynaBean"/>
</form-beans>
```

As you can see, it is very simple and fast to set up a Lazy DynaBean, saving much of the time traditionally spent in explicitly declaring the fields a Form Bean has.

Indexed and Mapped Properties

Using simple properties and nested properties on Form Beans will satisfy most requirements; however, there are scenarios where it is necessary to use a collection as a property. For example, when creating a form that has a variable number of fields, a collection must be used to capture the field data because the exact number of fields is not known in advance. Collection properties are also useful for transferring and displaying a variable-length list of data returned from the Model layer of an application. Struts supports two types of collection properties: indexed properties (e.g., arrays and **java.util.List** descendants such as **ArrayList**) and mapped properties (e.g., **java.util.Map** descendants such as **HashMap**).

Indexed properties are those properties that are backed by an indexed collection, such as arrays, **ArrayList**s, etc. To use an indexed property, you must set up the proper type of getter and setter methods in your Form Bean, as shown here:

```
import org.apache.struts.action.ActionForm;


public class EmployeeForm extends ActionForm
{
  private String[] departments =
    {"Accounting", "Sales", "Marketing", "IT'};


  public String getDepartments(int index) {
    return departments[index];
  }
  public void setDepartments(int index, String value) {
    departments[index] = value;
```

```
  }
}
```

This example uses an array-based indexed property called **departments**. Notice that the **getDepartments( )** and **setDepartments( )** methods take an **index** argument to specify the index in the array for the value that should be gotten or set. **java.util.List**-based indexed properties only require you to create a getter method, as shown next:

```
import java.util.ArrayList;
import java.util.List;
import org.apache.struts.action.ActionForm;

public class EmployeeForm extends ActionForm
{
  private ArrayList departments = new ArrayList();

  public EmployeeForm() {
    departments.add("Accounting");
    departments.add("Sales");
    departments.add("Marketing");
    departments.add("IT");
  }

  public List getDepartments() {
    return departments;
  }
}
```

Struts takes care of calling the getter and setter methods for indexed properties with the proper index when capturing data from a form or when populating a form. The following example illustrates how to reference an indexed property using the Struts tag libraries:

```
<html:form>
  <html:text property="departments[0]"/>
</html:form>
```

Notice that an index is specified for the **departments** field using **[ ]** notation. When this hypothetical form is submitted, an index of 0 and the value entered in the control will be used to populate the proper element in the collection. While this example illustrates how to reference a specific element in an indexed property, the real power of indexed properties is realized when a loop is used so that individual element indexes don't have to be specified. The following example illustrates how to use a loop to iterate over an indexed property's elements.

```
<html:form>
  <logic:iterate name="employeeForm" property="departments"
                 id="department" indexId="index">
```

```
    <html:text property="<%="departments[" + index + "]"%>"/>
  </logic:iterate>
</html:form>
```

A variable number of text input fields will be generated. If the **departments** property has 10 elements, 10 text input fields will be generated; if **departments** has no elements, no text input fields will be generated.

Mapped properties work much the same way that indexed properties do—only they are backed by **java.util.Map** descendants instead of arrays or **java.util.List** descendants. The syntax for referencing elements in a map differs as well. Instead of using **[ ]** notation to specify an index, **( )** notation is used to specify a key in the map. An example Form Bean with a mapped property is shown here.

```
import java.util.HashMap;
import java.util.Map;
import org.apache.struts.action.ActionForm;

public class EmployeeForm extends ActionForm
{
  private HashMap departments = new HashMap();

  public EmployeeForm() {
    departments.put("dep1", "Accounting");
    departments.put("dep2", "Sales");
    departments.put("dep3", "Marketing");
    departments.put("dep4", "IT");
  }

  public Object getDepartments(String key) {
    return departments.get(key);
  }

  public void setDepartments(String key, Object value) {
    departments.put(key, value);
  }
}
```

The **getDepartments( )** and **setDepartments( )** methods take a key argument to specify the key in the map for the value that should be gotten or set. Referencing mapped properties is similar to referencing indexed properties, as shown here:

```
<html:form>
  <html:text property="departments(dep1)"/>
</html:form>
```

Note Indexed and mapped properties can be both nested and contain nested properties, just as any other type of property can be.

JSP Tag Libraries

Struts comes packaged with a set of its own custom JSP tag libraries that aid in the development of JSPs. The tag libraries are fundamental building blocks in Struts applications because they provide a convenient mechanism for creating HTML forms whose data will be captured in Form Beans and for displaying data stored in Form Beans. Additionally, the Struts tag libraries provide several utility tags to accomplish things such as conditional logic, iterating over collections, and so on. However, with the advent of the JSP Standard Tag Library (JSTL), many of the utility tags have been superceded. (Using JSTL with Struts is covered in Chapter 17.)

Following is a list of the Struts tag libraries and their purpose:

- **HTML** Used to generate HTML forms that interact with the Struts APIs.
- **Bean** Used to work with Java bean objects in JSPs, such as to access bean values.
- **Logic** Used to cleanly implement simple conditional logic in JSPs.
- **Nested** Used to simplify access to arbitrary levels of nested objects from the HTML, Bean, and Logic tags.

Later in this book, each of these libraries has an entire chapter dedicated to its use, but this section provides a brief introduction to using the tag libraries, focusing on the core Struts JSP tag library, the HTML Tag Library, as an example. This library is used to generate HTML forms that, when submitted, populate Form Beans. Additionally, the HTML Tag Library tags can create HTML forms populated with data from Form Beans. To use the HTML Tag Library in a Struts application, your application's JSPs must declare their use of the library with a JSP **taglib** directive:

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
```

Notice that the **prefix** attribute is set to "html". This attribute can be set to whatever you want; however, "html" is the accepted default for the HTML Tag Library. The **prefix** attribute declares the prefix that each tag must have when it is used in the JSP, as shown here:

```
<html:form action="/logon">
```

Because "html" was defined as the prefix, the **form** tag was used as shown. However, if you chose to use a prefix of "strutshtml", the tag would be used the following way:

```
<strutshtml:form action="/logon">
```

Note Modern application servers use the **uri** attribute of the **taglib** directive to automatically resolve the location of the tag library descriptor file. Older application servers that support only JSP version 1.1 and/or version 1.0 require that tag libraries be registered in the **web.xml** file so that they can be resolved, as shown here:

```
<taglib>
  <taglib-uri>http://struts.apache.org/tags-html</taglib-uri>
  <taglib-location>/WEB-INF/tlds/struts-html.tld</taglib-location>
</taglib>
```

Resource Bundles

Resource bundles allow Java applications to be easily internationalized by having application content placed into bundles. This content can then be read by the application at run time. Therefore, instead of having content hard-coded in the application, the application reads its content from the bundle. A

side benefit of using resource bundles to store application content (whether for internationalization or not) is that the content can be changed without having to recompile the application. Additionally, bundles serve as a central repository for content that is common to multiple uses (i.e., multiple applications). Having content in a central repository reduces unnecessary duplication.

Struts has built-in support for working with Java's resource bundle mechanism. Having this support allows the Struts framework to seamlessly support application internationalization as well as have a mechanism for externalizing content so that it can be easily changed without having to modify JSPs or application code. Struts uses resource bundle resources throughout the framework. For example, resource bundle resources can be accessed from JSPs to populate them with content. Similarly, action objects can access content stored in resource bundles to do such things as generate error or informational messages that get displayed on screen. The Struts Form Bean validation mechanism is also tied to resource bundles for managing error messages. Actually, there are several uses for resource bundles throughout Struts.

The rest of this section explains how to create a resource bundle properties file and configure Struts to use it. An example of accessing resource bundle content from a JSP is also shown. Later chapters provide specific information about how to use resource bundles in the context of those chapters' topics.

Using resource bundles in Struts is as easy as creating a properties file to store the resources in, and then configuring Struts to use the properties file. Once this is done, accessing the resources is straightforward. Following is a very simple resource bundle properties file containing a few properties:

```
page.title=Employee Search
link.employeeSearch=Search for Employees
link.addEmployee=Add a New Employee
```

Resource bundle properties files simply contain key/value pairs. The resources are accessed by their key. The standard name for the resource bundle properties file in Struts is**MessageResources.properties**. In order for Struts to be able to load this file, it must be stored on your application's classpath. For example, it could be stored in the **/WEB-INF/classes**directory.

The following snippet configures the resource bundle with Struts:

```
<!-- Message Resources Configuration -->
<message-resources
  parameter="com.jamesholmes.minihr.MessageResources"/>
```

The **parameter** attribute of the **message-resources** tag specifies the fully qualified name of the resource bundle properties file minus the **.properties** file extension. In this example, a file named **MessageResources.properties** would be stored in the **/WEB-INF/classes/com/jamesholmes/minihr** directory.

Once a properties file has been created and configured in the Struts configuration file, the resources in the bundle can be accessed from several places in the Struts framework. The most common place is in JSPs. The following snippet illustrates how to use the Bean Tag Library's **message** tag to load a message from the resource bundle:

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>


<html>
<head>
```

```
<title><bean:message key="page.title"/></title>
</head>
<body>

...
```

The value specified with the **message** tag's **key** attribute is the key for a message in the resource bundle. At run time, Struts retrieves the message and places it in the JSP.

Note Detailed information on resource bundles and internationalizing Struts applications is found in Chapter 10.

Reviewing the View Layer of the Mini HR Application

To solidify your understanding of the View layer of Struts applications, it will be helpful to review the View layer of the Mini HR application developed in Chapter 2. Doing so clearly illustrates the core components involved in creating the View layer.

Mini HR's View layer consists of a Form Bean, two JSPs, and a resource bundle properties file. The **SearchForm** Form Bean is shown next:

```
package com.jamesholmes.minihr;

import java.util.List;

import javax.servlet.http.HttpServletRequest;

import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.action.ActionMessage;

public class SearchForm extends ActionForm
{
  private String name = null;
  private String ssNum = null;
  private List results = null;

  public void setName(String name) {
    this.name = name;
  }

  public String getName() {
    return name;
  }
```

```
public void setSsNum(String ssNum) {
  this.ssNum = ssNum;
}

public String getSsNum() {
  return ssNum;
}

public void setResults(List results) {
  this.results = results;
}

public List getResults() {
  return results;
}

// Reset form fields.
public void reset(ActionMapping mapping, HttpServletRequest request)

{
  name = null;
  ssNum = null;
   results = null;
}

// Validate form data.
public ActionErrors validate(ActionMapping mapping,
  HttpServletRequest request)
{
  ActionErrors errors = new ActionErrors();

  boolean nameEntered = false;
  boolean ssNumEntered = false;

  // Determine if name has been entered.
  if (name != null && name.length() > 0) {
      nameEntered = true;
  }
```

```
      // Determine if social security number has been entered.
  if (ssNum != null && ssNum.length() > 0) {
    ssNumEntered = true;
  }

  /* Validate that either name or social security number
     has been entered. */
  if (!nameEntered && !ssNumEntered) {
    errors.add(null,
          new ActionMessage("error.search.criteria.missing"));
  }

  /* Validate format of social security number if
       it has been entered. */
  if (ssNumEntered && !isValidSsNum(ssNum.trim())) {
      errors.add("ssNum",
      new ActionMessage("error.search.ssNum.invalid"));
  }

  return errors;
}

// Validate format of social security number.
private static boolean isValidSsNum(String ssNum) {
    if (ssNum.length() < 11) {
     return false;
  }

  for (int i = 0; i < 11; i++) {
    if (i == 3 || i == 6) {
      if (ssNum.charAt(i) != '-') {
        return false;
      }
    } else if ("0123456789".indexOf(ssNum.charAt(i)) == -1) {
      return false;
    }
  }
```

```
    return true;
  }
}
```

The **SearchForm** class is a basic Form Bean with a few properties and implementations for the **reset( )** and **validate( )** method hooks. Mini HR uses this Form Bean to capture search criteria from the search page using the **name** and **ssNum** fields. The **results** field is used to transfer search results back to the search page after a search has been performed.

The **index.jsp** page, shown next, is a simple JSP used as an example menu page for linking to Mini HR's functions:

```
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>

<html>
<head>
<title>ABC, Inc. Human Resources Portal</title>
</head>
<body>

<font size="+1">ABC, Inc. Human Resources Portal</font><br>
<hr width="100%" noshade="true">

&#149; Add an Employee<br>
&#149; <html:link forward="search">Search for Employees</html:link><br>

</body>
</html>
```

This page is the opening page for the Mini HR application and provides a link to the Mini HR search page.

The **search.jsp** page shown here provides the core interface to the Mini HR search functionality. It serves as the search criteria page as well as the search results page.

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic" %>

<html>
<head>
<title>ABC, Inc. Human Resources Portal - Employee Search</title>
</head>
<body>
```

```
<font size="+1">
ABC, Inc. Human Resources Portal - Employee Search
</font><br>
<hr width="100%" noshade="true">

<html:errors/>

<html:form action="/search">

<table>
<tr>
<td align="right"><bean:message key="label.search.name"/>:</td>
<td><html:text property="name"/></td>
</tr>
<tr>
<td></td>
<td>-- or --</td>
</tr>
<tr>
<td align="right"><bean:message key="label.search.ssNum"/>:</td>
<td><html:text property="ssNum"/> (xxx-xx-xxxx)</td>
</tr>
<tr>
<td></td>
<td><html:submit/></td>
</tr>
</table>

</html:form>

<logic:present name="searchForm" property="results">

<hr width="100%" size="1" noshade="true">

<bean:size id="size" name="searchForm" property="results"/>
<logic:equal name="size" value="0">
<center><font color="red"><cTypeface:Bold>No Employees
Found</b></font></center>
</logic:equal>
```

```
<logic:greaterThan name="size" value="0">
<table border="1">
<tr>
<th>Name</th>
<th>Social Security Number</th>
</tr>
<logic:iterate id="result" name="searchForm" property="results">
<tr>
<td><bean:write name="result" property="name"/></td>
<td><bean:write name="result" property="ssNum"/></td>
</tr>
</logic:iterate>
</table>
</logic:greaterThan>

</logic:present>

</body>
</html>
```

This page uses Struts tag library tags to determine whether or not it is being executed before or after a search has been submitted. If the page is being displayed before a search, it simply displays the search criteria form. However, if the page is being displayed after a search, it displays the search criteria form in addition to the search results.

The **MessageResources.properties** resource bundle file, shown next, is used by the **SearchForm** Form Bean and the JSPs to retrieve externalized content from a central repository:

```
# Label Resources
label.search.name=Name
label.search.ssNum=Social Security Number

# Error Resources
error.search.criteria.missing=Search Criteria Missing
error.search.ssNum.invalid=Invalid Social Security Number
errors.header=<font color="red"><cTypeface:Bold>Validation
Error(s)</b></font><ul>
errors.footer=</ul><hr width="100%" size="1" noshade="true">
errors.prefix=<li>
errors.suffix=</li>
```

The **SearchForm** Form Bean uses this file to store validation error messages. The JSPs use this file to store field labels. Together they are able to leverage Struts' built-in resource bundle mechanism for externalizing content. Doing so allows for easy updates to the content and provides a simple interface for internationalizing the content if necessary.

Alternative View Technologies

Before concluding this chapter, it's important to point out again that Struts is an extensible Web framework that allows you to use any technology you desire to develop the View layer of your Struts application. While this chapter has focused primarily on the built-in JSP-related support, there are several alternatives available. The following table lists a few of the options and a brief description, including a URL for each option.

Technology	Description and URL
Cocoon	The Apache Cocoon project is an XML-based Web application publishing engine for generating, transforming, processing, and outputting data. Don Brown created a Struts plugin, Cocoon Plugin, which integrates the Cocoon framework with the Struts framework. http://struts.sourceforge.net/struts-cocoon/
stxx	Struts for transforming XML with XSL (stxx) is a third-party extension of the Struts framework that supports using XSLT (XML Style Language Templates) for the View layer. http://stxx.sourceforge.net/
Swing	Java's Swing library can be used to create rich GUI front ends for Struts applications. The following article illustrates how to do this. http://javaboutique.internet.com/tutorials/Swing/
Velocity	The Jakarta Velocity project is a Java-based templating engine that can be used as an alternative to JSPs. VelocityStruts is a Velocity subproject that integrates Velocity with the Struts framework. http://jakarta.apache.org/velocity/tools/struts/

Chapter 5: The Controller Layer

Overview

The Controller layer of an MVC application is responsible for creating the abstraction between the Model and View layers. That is, the Controller layer acts as a liaison between the Model and View layers, separating one from the other. This abstraction is the foundation of the MVC design pattern. Recall that the Model contains the application's core, including the business logic and data access code, and the View contains the interface to the application. The Controller layer stands between these two layers, allowing them to change independently of one another. With this architecture, the Model (or core application code) is not limited to a singular use. This is a key advantage of MVC.

In MVC Web applications, the Controller layer serves as the central point of access to the application. All requests to an MVC Web application flow through the controller. By doing so, the Controller layer provides processing common to all requests, such as security, caching, logging, and so on. Most importantly, though, having all requests flow through the Controller allows the Controller to have complete autonomy over how requests are mapped to business processing and how the proper View is selected, based on the outcome of the business processing.

Struts and the Controller Layer

Struts provides a robust Controller layer implementation that has been designed from the ground up to be extensible. At its core is the Controller servlet, **ActionServlet**, which is responsible for initializing a Struts application's configuration from the Struts configuration file and for receiving all incoming requests to the application. Upon receiving a request, **ActionServlet** delegates its processing to the Struts request processing engine. The request processing engine processes all aspects of the request, including selecting the Form Bean associated with the request, populating the Form Bean with data, validating the Form Bean, and then selecting the correct **Action** class to execute for the request. The **Action** class is where the Struts framework ends and your application code begins. **Action** classes provide the glue between the View and Model layers and return instances of the **ActionForward** class to direct the Controller which View to display. Figure 5-1 illustrates the Controller layer lifecycle.

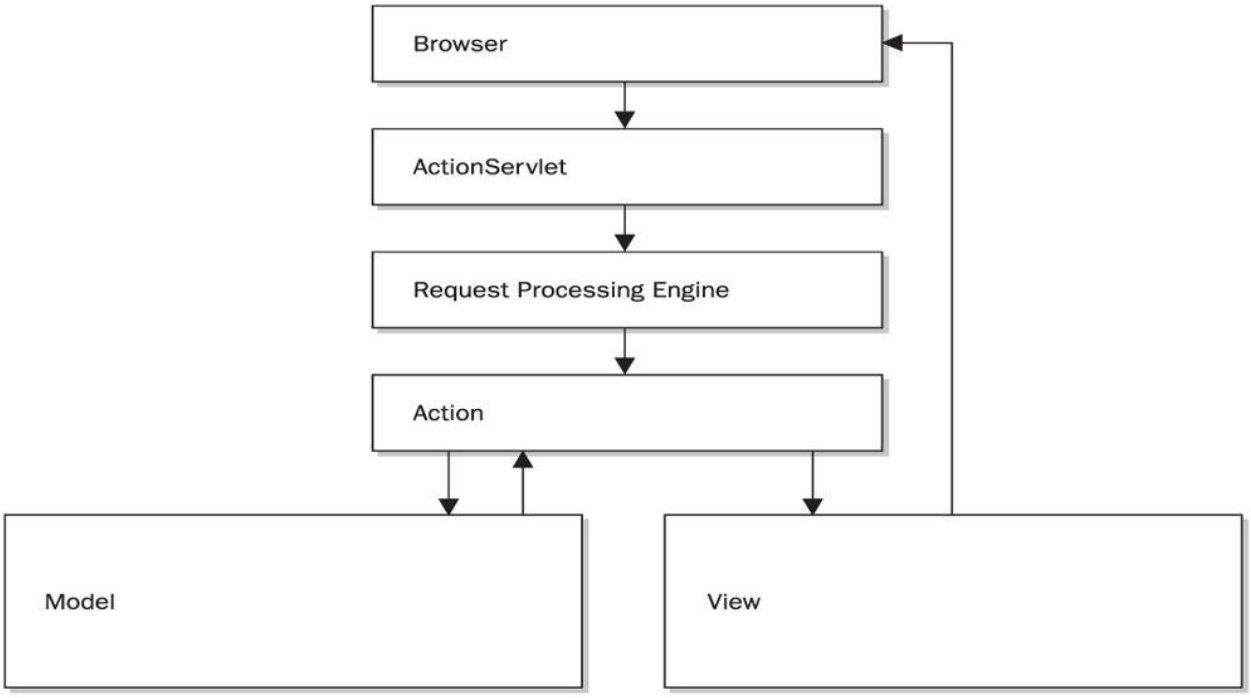


Figure 5-1: The Controller layer lifecycle

The following sections explain each of the major Controller layer components in detail.

The ActionServlet Class

The **ActionServlet** class is the main controller class that receives all incoming HTTP requests for the application. Additionally, **ActionServlet** is responsible for initializing the Struts framework for your application. Like any other servlet, **ActionServlet** must be configured in your application's Web application deployment descriptor: **web.xml**. The configuration settings in **web.xml**specify how to map requests to your Web application to the **ActionServlet**. There are two ways that **ActionServlet** can be configured to receive requests in **web.xml**. First, **ActionServlet**can be configured using path mapping, as shown here:

```
<?xml version="1.0"?>


<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
```

```
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <!-- Action Servlet Configuration -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <!-- Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>/do/*</url-pattern>
  </servlet-mapping>

</web-app>
```

Path mapping routes to **ActionServlet** all requests that match a specified path. The default path is **/do/***, as shown in the preceding **web.xml** file; however, you can use any path that you like.

The second way to map requests to **ActionServlet** is to use extension mapping, as shown next:

```
<?xml version="1.0"?>

<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app>

  <!-- Action Servlet Configuration -->
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>config</param-name>
```

```
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <!-- Action Servlet Mapping -->
  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>

</web-app>
```

Extension mapping maps to **ActionServlet** all requests with the specified extension. The default extension to use is **.do;** however, you can use any extension you like.

Note Extension mapping is required if you are using Struts' module feature. For more information on modules, see Chapter 9.

The Request Processing Engine

Struts uses its request processing engine to perform the processing for all requests received by the **ActionServlet**. The request processing engine takes each request and breaks its processing down into several small tasks. This approach allows each individual part of the request processing cycle to be customized.

In version 1.3, a new request processing engine was introduced that is built on top of the Jakarta Commons Chain library (http://jakarta.apache.org/commons/chain/). Previous versions of Struts performed all of the request processing in a single class named **org.apache.struts.action.RequestProcessor**. Both of these request processing methods are described separately in the following sections.

Jakarta Commons Chain-Based Request Processing

The **RequestProcessor** class-based processing originally used in Struts (and described in the following eponymously named section) provided a nice abstraction for each part of the request processing cycle; however, it had its limitations. The principal limitation was that customization of the request processing cycle required subclassing the **RequestProcessor** class. This subclassing precluded multiple independent customizations to be created and used together because there was not a way to tie the assorted customizations back together. Because of this, Commons Chain is now being utilized by Struts to further abstract the processing of requests and solve the problem of having multiple customizations.

The Commons Chain library provides an implementation of the *Chain of Responsibility (COR)* pattern described in the seminal **Design Patterns** book by the *Gang of Four.* The COR pattern models a computation as a series of *commands* that are combined into a *chain.* Chains represent the entire computation, while commands represent discrete pieces of the computation. Each command in a chain is executed in succession until all commands in the chain have been executed or until a command returns a flag indicating that execution is complete for the chain. This pattern of processing is essentially what was implemented in the Struts **RequestProcessor** class. However,

the **RequestProcessor** class did not provide a mechanism for introducing new commands into (or removing commands from) its chain. Additionally, the list of steps and the order of the steps of processing in the **RequestProcessor** class is hard-coded and cannot be changed without subclassing **RequestProcessor**. Commons Chain allows commands to be easily added or removed by abstracting the definition of chains into an XML configuration file.

In converting to Commons Chain, each of the processing methods of the **RequestProcessor** class was moved to its own command class and all of the commands were wired together into a chain. The chain definition is stored in the default Struts Chain configuration file, **chain-config.xml**, that is stored in the Struts Core **.jar** file (e.g., **struts-core-1.3.5.jar**) in the**org\apache\struts\chain** directory. To add or remove commands from the default chain configuration, the Chain configuration file must be modified. Because the Chain configuration file is stored in the Struts Core **.jar** file, it is cumbersome to update that file. Instead, a new Chain configuration file should be created. The default Chain configuration file can be used as a template and then any changes can be made. Once the new Chain configuration file is created, Struts must be configured to use it as shown in the following example:

```
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
  <init-param>
    <param-name>config</param-name>
    <param-value>/WEB-INF/struts-config.xml</param-value>
  </init-param>
  <init-param>
    <param-name>chainConfig</param-name>
    <param-value>/WEB-INF/chain-config.xml</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Like the Struts configuration file, the Chain configuration file is specified with an initialization parameter to the Struts **ActionServlet**. The initialization parameter is named **chainConfig** and its value is the path to the Chain configuration file.

RequestProcessor Class-Based Processing

As stated, in versions of Struts prior to version 1.3, the **RequestProcessor** class was used to handle all request processing. The **RequestProcessor** class uses a separate method to carry out each request processing task. Each of the separate methods is aptly named with a prefix of *process;* for example, **processMultipart( )** and **processPath( )**.

Table 5-1 lists and describes briefly each of the **process*( )** methods from the **RequestProcessor** class (in the order they are executed).

Table 5-1: The process*( ) Methods of the RequestProcessor Class

Method	Description
processMultipart( )	Wraps multipart requests with a special wrapper class.

Table 5-1: The process*( ) Methods of the RequestProcessor Class

Method	Description
processPath( )	Determines the path that will be used to select an action to which processing is delegated.
processLocale( )	Saves the user's locale in session scope.
processContent( )	Sets the default content type for the response.
processNoCache( )	Sets no-cache HTTP headers for the response if necessary.
processPreprocess( )	Provides a hook for subclasses to override. It is used to tell the request processor whether or not to continue processing the request after this method has been called.
processCachedMessages( )	Removes cached **ActionMessage** objects from the session so that they are available only for one request.
processMapping( )	Selects the action mapping to use for the request.
processRoles( )	Checks if the current user has a role that is allowed to access the requested resource.
processActionForm( )	Creates a new Form Bean or retrieves one from the session for the request.
processPopulate( )	Populates the Form Bean returned from **processActionForm( )** with data from the incoming request.
processValidate( )	Invokes the **validate( )** method on the Form Bean returned from **processActionForm( )** if necessary.
processForward( )	Processes the forward for the action mapping matching the current request path, if the matching mapping is specified to be a forward.
processInclude( )	Processes the include for the action mapping matching the current request path, if the matching mapping is specified to be an include.
processActionCreate( )	Creates or recycles an existing action to process the current request.
processActionPerform( )	Invokes the **execute( )** method on the action returned from **processActionCreate( )**.
processForwardConfig( )	Forwards to the forward returned from **processActionPerform( )**.

By having each phase of the request processing cycle take place in a separate method, request processing can easily be customized. Simply create a custom request processor that extends the base **RequestProcessor** class and override the methods that need to be customized. For example, a custom request processor can apply a logged-in security check before any action is executed. The **RequestProcessor** class provides the **processPreprocess( )** method hook expressly for this. The **processPreprocess( )** method is called before actions are executed. The following example shows how to do this:

```
package com.jamesholmes.minihr;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.RequestProcessor;

public class LoggedInRequestProcessor extends RequestProcessor
{
  protected boolean processPreprocess(
    HttpServletRequest request,
    HttpServletResponse response)
  {
    // Check if user is logged in.
    // If so return true to continue processing,
    // otherwise return false to not continue processing.
    return (true);
  }
}
```

To use a custom request processor, you have to configure Struts to use it in the Struts configuration file:

```
<controller
processorClass="com.jamesholmes.minihr.LoggedInRequestProcessor"/>
```

Note When using the Struts module feature, each module has its own request processor. Thus, if you want to apply a custom request processor to all modules, you must configure it in each module's Struts configuration file.

The Action Class

The **Action** class is where the Struts framework ends and your application code begins. As previously mentioned, **Action** classes provide the glue between the View and Model layers and are responsible for processing specific requests. **Action** classes are intended to transfer data from the View layer to a specific business process in the Model layer, and then to return data from the business process to the View layer. Business logic should not be embedded in actions, because that violates the principles of MVC.

Each action is mapped to a path in the Struts configuration file. When a request with the specified path is made to the **ActionServlet**, the action is invoked to process the request. The following snippet illustrates how to configure an action in the Struts configuration file:

```
<action-mappings>
  <action path="/UpdateUser"
          type="com.jamesholmes.example.UpdateUserAction"/>
</action-mappings>
```

Which URL is used to access the action depends on how **ActionServlet** is configured in **web.xml**. If **ActionServlet** is configured to use path mapping, the action defined in the preceding example is accessed as **http://localhost:8080/MiniHR/do/UpdateUser**, assuming a server of **localhost** running on port **8080** and an application deployed as **MiniHR**. If extension mapping were used to configure **ActionServlet**, the URL would be **http://localhost:8080/MiniHR/UpdateUser.do**.

When an action is called to process a request, its **execute( )** method is invoked. The **execute( )** method is analogous to the **service( )** method in servlets. It handles all processing. Following is an example **Action** subclass and its **execute( )** method:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class UpdateUserAction extends Action
{
  public ActionForward execute(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Perform request processing here.
  }
}
```

Retrieving Values from Form Beans

Action classes are the principal location for transferring data from the View layer to the Model layer and vice versa. Form Beans are used to encapsulate the data being transferred to and from the View layer and are passed to the **execute( )** method of **Action** classes. **Action** classes then retrieve values from the Form Bean and transfer the values to the Model Layer. Form Beans should not be transferred directly to the Model layer because it creates an artificial dependency on Struts. Because of this, it is important to understand how to retrieve the data stored in Form Beans. Each of the different types of Form Beans has a different mechanism for retrieving the values stored in the Form Bean and is explained here.

Standard Form Beans that subclass **org.apache.struts.action.ActionForm** have basic getter and setter methods that are called directly to retrieve the values from the Form Bean as shown here:

```
public ActionForward execute(ActionMapping mapping,
  ActionForm form,
  HttpServletRequest request,
```

```
  HttpServletResponse response)
  throws Exception
{
  SearchForm searchForm = (SearchForm) form;

  String name = searchForm.getName();
  String ssNum = searchForm.getSsNum();
}
```

Notice that the incoming Form Bean must be cast to its proper type before calling any getter or setter methods.

The following code sample illustrates how to access the properties of a dynamic Form Bean. Dynamic Form Beans do not have concrete getter and setter methods. Instead their values are retrieved by passing the name of a field to the **get( )** method of the **DynaActionForm** class.

```
public ActionForward execute(ActionMapping mapping,
  ActionForm form,
  HttpServletRequest request,
  HttpServletResponse response)
  throws Exception
{
  DynaActionForm searchForm = (DynaActionForm) form;

  String name = (String) searchForm.get("name");
  String ssNum = (String) searchForm.get("ssNum");
}
```

The **DynaActionForm get( )** method has a return type of **Object** thus fields must be cast to their proper type when being retrieved.

Lazy DynaBeans operate identically to Dynamic Form Beans: simply pass the name of the field being retrieved to the **get( )** method of the **LazyDynaBean** class.

```
public ActionForward execute(ActionMapping mapping,
  ActionForm form,
  HttpServletRequest request,
  HttpServletResponse response)
  throws Exception
{
  LazyDynaBean searchForm = (LazyDynaBean) form;

  String name = (String) searchForm.get("name");
  String ssNum = (String) searchForm.get("ssNum");
}
```

Customizing the Response from an Action

By default, an **ActionForward** should be returned from the **execute( )** method of actions to direct the controller on which view should be displayed. The controller takes care of routing to the proper JSP and the JSP generates the response to the browser. There are scenarios, however, where it is necessary to customize the response generated from an action. File downloads are an example of this. Instead of forwarding to a JSP to render a page, an action can interface directly with the HTTP response and transmit the file being downloaded. To do this, an action simply has to use the **HttpServletResponse** object passed to **Action** classes' execute **method( )**. Additionally the action must return **null** from the **execute( )** method to indicate to the Struts Controller that no further processing is required. Following is an example file download action:

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class DownloadAction extends Action
{
  public ActionForward execute(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Perform file download processing here.

    return null;
  }
}
```

Notice that **null** is returned from the **Action** class instead of an **ActionForward**. This is a critical detail when taking control of the HTTP response directly from within the **Action** class.

Struts' Built-in Actions

Struts comes packaged with several built-in utility actions that provide functionality that is useful to many applications. Table 5-2 lists each of the built-in actions and their purpose.

Table 5-2: Struts' Built-in Utility Actions

Action	Description
DispatchAction	Provides a mechanism for modularizing a set of related functions into a single action, thus eliminating the need to create separate, independent actions for each function.
DownloadAction	Provides a mechanism for easing the creation of file download actions.
EventDispatchAction	Provides a mechanism for modularizing a set of related functions into a single action, thus eliminating the need to create separate, independent actions for each function.
ForwardAction	Provides a mechanism for forwarding to a specified URL.
IncludeAction	Provides a mechanism for including the contents of a specified URL.
LocaleAction	Provides a mechanism for setting a user's locale and then forwarding to a specified page.
LookupDispatchAction	Provides a mechanism for modularizing a set of related functions into a single action, thus eliminating the need to create separate, independent actions for each function.
MappingDispatchAction	Provides a mechanism for modularizing a set of related functions into a single action, thus eliminating the need to create separate, independent actions for each function.
SwitchAction	Provides a mechanism for switching between modules in a modularized Struts application.

The following sections describe each of the built-in actions in detail.

The DispatchAction Class

The **org.apache.struts.actions.DispatchAction** class provides a mechanism for modularizing a set of related functions into a single action, thus eliminating the need to create separate, independent actions for each function. For example, consider a set of related functions for adding a user, updating a user, and removing a user. Instead of creating an **AddUserAction** class, an**UpdateUserAction** class, and a **RemoveUserAction** class, by extending **DispatchAction**, you can create one **UserAction** class that has three methods: **add( )**, **update( )**, and **remove( )**. At run time, **DispatchAction** manages routing requests to the appropriate method in its subclass. **DispatchAction** determines which method to call based on the value of a request parameter that is passed to it from the incoming request.

To use **DispatchAction**, you must create a subclass from it and provide a set of methods that will be called to process requests. Additionally, you have to set up for the action an action mapping that specifies the name of a request parameter that will be used to select which method should be called for each request. Following is an example **UserAction** class that extends**DispatchAction**:

```
package com.jamesholmes.minihr;


import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.DispatchAction;

public class UserAction extends DispatchAction
{
  public ActionForward add(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Add user.
    ...

    return mapping.findForward("success");
  }

  public ActionForward update(ActionMapping mapping,
     ActionForm form,
     HttpServletRequest request,
     HttpServletResponse response)
     throws Exception
  {
     // Update user.
     ...

     return mapping.findForward("success");
  }

  public ActionForward remove(ActionMapping mapping,
     ActionForm form,
     HttpServletRequest request,
     HttpServletResponse response)
     throws Exception
  {
     // Remove user.
     ...
```

```
      return mapping.findForward("success");
    }
}
```

Notice that this class does not provide an implementation for the **execute( )** method the way typical **Action** classes do. This is because **DispatchAction** provides to you an implementation of this method that manages delegating to the individual methods. In order for your **DispatchAction** subclass to work, you must create in the Struts configuration file an action mapping that specifies the name of a request parameter that will be used to select the method that will be called for specific requests. Following is a sample snippet that illustrates how to do this:

```
<action-mappings>
  <action path="/User"
          type="com.jamesholmes.minihr.UserAction"
      parameter="function"/>
</action-mappings>
```

The value specified with the **parameter** attribute of the **action** tag will be used as the name of a request parameter that will contain the name of a method to invoke for handling the request. Given the preceding mapping of **/User** to **UserAction**, the following URL will invoke the **add( )** method (assuming the application was run on your **localhost** at port **8080** and the application was deployed as **/MiniHR/**):

http://localhost:8080/MiniHR/User.do?function=add

To invoke the **remove( )** method, use the following URL:

http://localhost:8080/MiniHR/User.do?function=remove

The DownloadAction Class

The **org.apache.struts.actions.DownloadAction** class provides a mechanism for easing the creation of file download actions. Instead of creating an action from scratch with all of the required infrastructure code for downloading files, the **DownloadAction** class can be extended. **DownloadAction** provides all of the infrastructure code in a simple, reusable package.**DownloadAction** subclasses must include a **getStreamInfo( )** method that returns a **StreamInfo** instance with details about the file to download.

```
package com.jamesholmes.minihr;

import java.io.File;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.DownloadAction;
```

```
public class ReportDownloadAction extends DownloadAction
{
  protected StreamInfo getStreamInfo(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    String contentType = "application/vnd.ms-excel";
    File file = new File("/dir1/dir2/report.xls");

    return new FileStreamInfo(contentType, file);
  }
}
```

When the **DownloadAction** subclass, such as **ReportDownloadAction** in the preceding example, is invoked, it will stream the file specified in the **getStreamInfo( )** method back to the browser.

The EventDispatchAction Class

The **org.apache.struts.actions.EventDispatchAction** class is a subclass of **DispatchAction** and provides a mechanism for modularizing a set of related functions into a single action, thus eliminating the need to create separate, independent actions for each function. For example, consider a set of related functions for adding a user, updating a user, and removing a user. Instead of creating an **AddUserAction** class, an **UpdateUserAction** class, and a **RemoveUserAction** class, by extending **EventDispatchAction**, you can create one **UserAction** class that has three methods: **add( )**, **update( )**, and **remove( )**. At run time, **EventDispatchAction** manages routing requests to the appropriate method in its subclass. **EventDispatchAction** determines which method to call based on the presence of a request parameter that is passed to it from the incoming request.

To use **EventDispatchAction**, you must create a subclass from it and provide a set of methods that will be called to process requests. Additionally, you have to set up for the action an action mapping that specifies the names of request parameters that will be used to select which method should be called for each request. Following is an example **UserAction** class that extends**EventDispatchAction**:

```
package com.jamesholmes.minihr;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.EventDispatchAction;
```

```
public class UserAction extends EventDispatchAction
{
  public ActionForward add(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Add user.
    ...

    return mapping.findForward("success");
  }

  public ActionForward update(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Update user.
    ...

    return mapping.findForward("success");
  }

  public ActionForward remove(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Remove user.
    ...

    return mapping.findForward("success");
  }
}
```

Notice that this class does not provide an implementation for the **execute( )** method the way typical **Action** classes do. This is because **EventDispatchAction** provides to you an implementation of this method that manages delegating to the individual methods. In order for your **EventDispatchAction** subclass to work, you must create in the Struts configuration file an action mapping that specifies the names of request parameters that will be used to select the method that will be called for specific requests. Following is a sample snippet that illustrates how to do this:

```
<action-mappings>
  <action path="/User"
          type="com.jamesholmes.minihr.UserAction"
     parameter="add,update,delete=remove,default=add"/>
</action-mappings>
```

The value specified with the **parameter** attribute of the **action** tag specifies a comma-delimited list of method names to dispatch to if a request parameter of the same name is present in the request. In the preceding example, the **add( )** method will be invoked for handling the request if an "add" request parameter is sent to the action. The value of the "add" request parameter is disregarded; only the presence of the request parameter is needed. The **remove( )** method will be invoked if a request parameter named "delete" is sent. The **remove( )** method, instead of a**delete( )** method, is invoked because of an alias of "delete". Method aliases are specified using *alias=method* notation, where *alias* is the name of the alias for a method and *method* is the name of the method to invoke. A default method can be specified by using an alias with the name of "default" as shown in the preceding example. Default aliases are used for scenarios where no request parameter matches the method names specified with the **action** tag's **parameter** attribute.

Given the preceding mapping of **/User** to **UserAction**, the following URL will invoke the **add( )** method (assuming the application was run on your **localhost** at port **8080** and the application was deployed as **/MiniHR/**):

http://localhost:8080/MiniHR/User.do?add.x=103

To invoke the **remove( )** method, use the following URL:

http://localhost:8080/MiniHR/User.do?remove=true

The ForwardAction Class

The **org.apache.struts.actions.ForwardAction** class provides a mechanism for forwarding to a specified URL. As explained earlier, in an MVC Web application, all requests to the application are supposed to flow through the Controller servlet. This ensures that the Controller layer of the application has an opportunity to prepare any resources that may be needed to handle the request (i.e., selecting the correct module and so on). **ForwardAction** is provided as a simple utility action that can be used for scenarios in which you simply want to link to a JSP page. Of course, linking directly to the JSP would be a violation of the MVC principles because all requests are supposed to be routed through the Controller. **ForwardAction** can be used to create links to JSPs so that you don't have to create an action whose only responsibility is to forward a request every time you want to link to a JSP. With **ForwardAction**, you simply create an action mapping in the Struts configuration file and specify the location to which the action will forward.

To use **ForwardAction**, simply create action mapping entries in the Struts configuration file, as shown next:

```
<action-mappings>
  <action path="/menu"
```

```
          type="org.apache.struts.actions.ForwardAction"
     parameter="/menu.jsp/>
</action-mappings>
```

For each page to which you want to link, you must create an action mapping. Each action mapping uses **ForwardAction**, but each specifies a different path for the action. The **parameter**attribute specifies the URL that will be forwarded to when the specified path is accessed.

An alternative solution to using **ForwardAction** is to use the **forward** attribute of the **action** tag in the Struts configuration file, as shown here:

```
<action-mappings>
  <action path="/menu"
       forward="/menu.jsp"/>
</action-mappings>
```

These two approaches effectively yield the same results.

The IncludeAction Class

The **org.apache.struts.actions.IncludeAction** class provides a mechanism for including the contents of a specified URL. This action behaves similarly to **ForwardAction**, but instead of forwarding to the specified URL, the specified URL is included. This action is useful when you want to include the contents of one page in another.

Using **IncludeAction** is quite easy. Just create action mapping entries in the Struts configuration file:

```
<action-mappings>
  <action path="/menu"
          type="org.apache.struts.actions.IncludeAction"
     parameter="/menu.jsp/>
</action-mappings>
```

For each page you want to include, you must create an action mapping. Each action mapping uses **IncludeAction**, but specifies a different path for the action. The **parameter** attribute specifies the URL that will be included when the specified path is accessed.

An alternative solution to using **IncludeAction** is to use the **include** attribute of the **action** tag in the Struts configuration file, as shown here:

```
<action-mappings>
  <action path="/menu"
       include="/menu.jsp"/>
</action-mappings>
```

These two approaches effectively yield the same results.

The LocaleAction Class

The **org.apache.struts.actions.LocaleAction** class provides a mechanism for setting a user's locale and then forwarding to a specified page. This action provides a convenient mechanism for changing a user's locale. For example, consider a site that is offered in English and Spanish versions. **LocaleAction** can be used to offer users a way to switch between the two languages without having to change their browser settings. With **LocaleAction** you simply create an action

mapping and then link to the action, specifying request parameters for which locale to switch to and a page to forward after the locale has been switched.

To use **LocaleAction**, you must create an action mapping entry for it in the Struts configuration file and then link to the action, specifying locale information and a page to forward to after the locale has been set. Following is an example of how to configure **LocaleAction** in the Struts configuration file:

```
<action-mappings>
  <action path="/SwitchLocale"
          type="org.apache.struts.actions.LocaleAction"/>
</action-mappings>
```

Once configured in the Struts configuration file, **LocaleAction** can be put to use. Simply create a link to the action and specify the locale settings that will be set and a page to forward to. Locale settings are specified with two request parameters: **language** and **country**. The page to forward to after setting the locale is specified with the **page** request parameter. The following URL illustrates how to use the request parameters:

http://localhost:8080/MiniHR/SwitchLocale.do?country=MX&language=es&page=/Menu.do

This example URL sets the country to **MX** (Mexico) and the language to **es** (Spanish). The **/Menu.do** page will be forwarded to after the new locale has been set.

The LookupDispatchAction Class

The **org.apache.struts.actions.LookupDispatchAction** class is a subclass of **DispatchAction** and provides a mechanism for modularizing a set of related functions into a single action, thus eliminating the need to create separate, independent actions for each function. For example, consider a set of related functions for adding a user, updating a user, and removing a user. Instead of creating an **AddUserAction** class, an **UpdateUserAction** class, and a **RemoveUserAction** class, by extending **LookupDispatchAction**, you can create one **UserAction** class that has three methods: **add( )**, **update( )**, and **remove( )**.

At run time, **LookupDispatchAction** manages routing requests to the appropriate method in its subclass. **LookupDispatchAction** determines which method to route to based on the value of a request parameter being passed to it from the incoming request. **LookupDispatchAction** uses the value of the request parameter to reverse-map to a property in the Struts resource bundle file (e.g., **MessageResources.properties**). That is, the value of the request parameter is compared against the values of properties in the resource bundle until a match is found. The key for the matching property is then used as a key to another map that maps to a method in your **LookupDispatchAction** subclass that will be executed.

To use **LookupDispatchAction**, you must create a subclass from it and provide a set of methods that will be called to process requests. The subclass must also include a**getKeyMethodMap( )** method that maps methods in the class to keys in the Struts resource bundle file. Additionally, you have to set up for the action an action mapping that specifies the name of a request parameter that will be used to select which method will be called for each request. Following is an example **UserAction** class that extends **LookupDispatchAction**:

```
package com.jamesholmes.minihr;


import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.LookupDispatchAction;

public class UserAction extends LookupDispatchAction
{
  protected Map getKeyMethodMap()
  {
    HashMap map = new HashMap();
    map.put("button.add", "add");
    map.put("button.update", "update");
    map.put("button.remove", "remove");

    return map;
  }

  public ActionForward add(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Add user.
    ...

    return mapping.findForward("success");
  }

  public ActionForward update(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Update user.
    ...
```

```
    return mapping.finwForward("success");
  }

  public ActionForward remove(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Remove user.
    ...

    return mapping.findForward("success");
  }
}
```

Notice that this class does not provide an implementation for the **execute( )** method as other **Action** classes do. This is because **LookupDispatchAction** provides to you an implementation of this method that manages delegating to the individual methods. Notice also the implementation of the **getKeyMethodMap( )** method. This method is required by **LookupDispatchAction**subclasses and is used to map the names of keys in the Struts resource bundle file to methods in the class. The keys' values in the bundle file are used to match against the value of the incoming request parameter specified by the **parameter** attribute of the **action** tag in the Struts configuration file.

In order for your **LookupDispatchAction** subclass to work, you must create in the Struts configuration file an action mapping that specifies the name of a request parameter that will be used to select the method that will be called for a specific request. Following is a sample snippet that illustrates how to do this:

```
<action-mappings>
  <action path="/User"
          type="com.jamesholmes.minihr.UserAction"
     parameter="function"/>
</action-mappings>
```

The value specified with the **parameter** attribute of the **action** tag will be used as the name of a request parameter that will contain the value of a key in the Struts resource bundle shown here:

```
button.add=Add User
button.update=Update User
button.remove=Remove User
```

LookupDispatchAction will use the value of the incoming request parameter to perform a reverse lookup for a key in the resource bundle. The matching key is then mapped to the appropriate method to execute based on the key-to-method mapping specified by the **getKeyMethodMap( )** method.

Given the preceding mapping of **/User** to **UserAction**, the following URL will invoke the **add( )** method (assuming the application was run on your **localhost** at port **8080** and the application was deployed as **/MiniHR/**):

http://localhost:8080/MiniHR/User.do?function=Add%20User

To invoke the **remove( )** method, use the following URL:

http://localhost:8080/MiniHR/User.do?function=Remove%20User

The MappingDispatchAction Class

The **org.apache.struts.actions.MappingDispatchAction** class is a subclass of **DispatchAction** and provides a mechanism for modularizing a set of related functions into a single action, eliminating the need to create separate, independent actions for each function. For example, consider a set of related functions for adding a user, updating a user, and removing a user. Instead of creating an **AddUserAction** class, an **UpdateUserAction** class, and a **RemoveUserAction** class, by extending **MappingDispatch Action**, you can create one **UserAction** class that has three methods: **add( )**, **update( )**, and **remove( )**. At run time, **MappingDispatchAction** manages routing requests to the appropriate method in its subclass. **MappingDispatchAction**determines which method to route based on the value of a parameter being passed to it from an action mapping in the Struts configuration file.

To use **MappingDispatchAction**, you must create a subclass from it and provide a set of methods that will be called to process requests. Additionally, you must set up action mappings that specify which method will be called for each request. Following is an example **UserAction** class that extends **MappingDispatchAction**:

```
package com.jamesholmes.minihr;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.apache.struts.actions.MappingDispatchAction;

public class UserAction extends MappingDispatchAction
{
  public ActionForward add(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Add user.
    ...
```

```
    return mapping.findForward("success");
  }

  public ActionForward update(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Update user.
    ...

    return mapping.findForward("success");
  }

  public ActionForward remove(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    // Remove user.
    ...

    return mapping.findForward("success");
  }
}
```

Notice that this class does not provide an implementation for the **execute( )** method as other **Action** classes do. This is because **MappingDispatchAction** provides to you an implementation of this method that manages delegating to the individual function methods.

In order for your **MappingDispatchAction** subclass to work, you must create in the Struts configuration file action mappings that specify the method that will be called for specific requests. Following is a sample snippet that illustrates how to do this:

```
<action-mappings>
  <action path="/AddUser"
          type="com.jamesholmes.minihr.UserAction"
     parameter="add"/>
  <action path="/UpdateUser"
```

```
          type="com.jamesholmes.minihr.UserAction"
      parameter="update"/>
  <action path="/RemoveUser"
          type="com.jamesholmes.minihr.UserAction"
      parameter="remove"/>
</action-mappings>
```

Notice that each action mapping uses the **UserAction** class, but specifies a different path for the action. Each of the unique paths will be processed by the same action, but a different method will be called based on the value specified with the **parameter** attribute. The value specified with the **parameter** attribute must match the name of a method in your **MappingDispatchAction**subclass.

The SwitchAction Class

The **org.apache.struts.actions.SwitchAction** class provides a mechanism for switching between modules in a modularized Struts application. As you'll see in Chapter 9, Struts enables you to modularize your Struts application. Each module has its own set of configuration data as well as its own request processor. **SwitchAction** works similarly to **ForwardAction**, except that before forwarding to a specified resource, the action changes the currently selected module. This is useful for forwarding to JSPs outside of the current module.

Note Detailed information on using the Struts modules feature is found in Chapter 9.

To use **SwitchAction**, you must create an action mapping entry for it in the Struts configuration file and then link to the action, specifying the module to switch to and a page to forward to after the module has been switched. Following is an example of how to configure **SwitchAction** in the Struts configuration file:

```
<action-mappings>
  <action path="/SwitchModule"
          type="org.apache.struts.actions.SwitchAction"/>
</action-mappings>
```

Once configured in the Struts configuration file, **SwitchAction** can be put to use. Simply create a link to the action and specify the module to switch to and a page to forward to afterward. The module to switch to is specified with the **prefix** parameter and the page to forward to afterward is specified with the **page** parameter. The following URL illustrates how to use the request parameters:
http://localhost:8080/MiniHR/SwitchModule.do?prefix=/Corporate&page=/Menu.do

This example URL switches to the **/Corporate** module, and the **/Menu.do** page will be forwarded to after the module has been switched.

The ActionForward Class

The **ActionForward** class encapsulates a *forward.* Forwards were introduced in Chapter 2, but this section takes a closer look at them because they are used by the **ActionForward** class.

Struts provides the forward as an alternative to hard-coding URLs inside your application. Forwards allow you to define logical names for URLs and then to use the names to reference the URLs. If you reference a URL by its logical name instead of referencing it directly, when the URL changes, you

don't have to update each reference to the URL. For example, with forwards you can define a forward for a search page with a logical name of "search" that points to the **/search.jsp** page. Instead of hard-coding the search page's **/search.jsp** URL throughout your application, you use the forward's logical name. If the location of the search page changes, you need to make only one change to the forward definition and all places in the application that point to that forward will receive the change. Essentially, forwards are URL aliases. They abstract URLs and shield you from changes. The application knows only the alias. Where the alias points does not matter.

Forwards are defined declaratively in the Struts configuration file. There are two types of forwards that can be defined, a global forward and an action-specific forward. Global forwards are available throughout an application, whereas action-specific forwards are available only to their respective action. Following is an example of how to define a global forward in the Struts configuration file:

```
<global-forwards>
  <forward name="searchPage" path="/search.jsp"/>
</global-forwards>
```

Action-specific forwards are defined by nesting the **forward** tag inside an **action** tag, as shown next:

```
<action-mappings>
  <action path="/updateUser"
          type="com.jamesholmes.minihr.UpdateUserAction">
    <forward name="success" path="/updateSuccess.jsp"/>
  </action>
</action-mappings>
```

Struts' **ActionForward** class encapsulates a forward inside an application. For example, the **Action** class's **execute( )** method has a return type of **ActionForward**. After an action executes, it must return an **ActionForward** or null (to indicate processing is complete). **ActionForward**s returned from actions are used to forward to the View layer. The following snippet illustrates how the **ActionForward** class is used in an action:

```
package com.jamesholmes.minihr;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class UpdateUserAction extends Action
{
  public ActionForward execute(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
```

```
    HttpServletResponse response)
    throws Exception
  {
    // Perform action processing.

    return new ActionForward("updateSuccess");
  }
}
```

The value passed to the **ActionForward** class's constructor corresponds to the logical name of a forward defined in the Struts configuration file.

Reviewing the Controller Layer of the Mini HR Application

To solidify your understanding of the Controller layer of Struts applications, this section reviews the Controller layer of the Mini HR application developed in Chapter 2. Doing so clearly illustrates the core components involved in creating the Controller layer.

Mini HR's Controller layer consists of a single class: **SearchAction**. The **SearchAction** class, shown next, is responsible for processing requests from the **search.jsp** page:

```
package com.jamesholmes.minihr;

import java.util.ArrayList;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public final class SearchAction extends Action
{
  public ActionForward execute(ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws Exception
  {
    EmployeeSearchService service = new EmployeeSearchService();
    ArrayList results;
```

```
    SearchForm searchForm = (SearchForm) form;

    // Perform employee search based on what criteria was entered.
    String name = searchForm.getName();
    if (name != null && name.trim().length() > 0) {
      results = service.searchByName(name);
    } else {
      results = service.searchBySsNum(searchForm.getSsNum().trim());
    }

    // Place search results in SearchForm for access by JSP.
    searchForm.setResults(results);

    // Forward control to this Action's input page.
    return mapping.getInputForward();
  }
}
```

When a search is initiated, the Struts **ActionServlet** Controller servlet delegates processing to this **Action** class. This class acts as the liaison between the Model layer and the View layer. Based on the search criteria entered from the View layer, **SearchAction** determines which search method to invoke on the **EmployeeSearchService** Model class and passes in the data from the View layer. **EmployeeSearchService** returns the results of the search and **SearchAction** manages getting the data back to the View layer.

Chapter 6: Validator

Overview

One of the major benefits of using the Struts framework is its built-in interface for performing data validations on incoming form data. As discussed inChapter 4, upon submitting an HTML form, Struts captures the form data and uses it to populate one of your application's **ActionForm** subclasses (Form Beans) assigned to the form. The Form Bean's **validate( )** method is then called to perform any necessary validation of the incoming data. If any validations fail, the HTML form is redisplayed so that the invalid data can be corrected. Otherwise, processing continues. This simple interface alleviates much of the headache associated with handling data validation, allowing you to focus on validation code and not the mechanics of capturing data and redisplaying incomplete or invalid data.

Struts' built-in validation interface, however, still has its shortcomings. Often, for example, validation code is heavily duplicated throughout an application because many fields require the same validation logic. Any change in the validation logic for similar fields requires code changes in several places, as well as recompilation of the affected code. To solve this problem and to enhance Struts' validation interface, David Winterfeldt created the Validator framework as a third-party add-on to Struts. Validator was later integrated into the core Struts code base and has since been detached from

Struts and is now a stand-alone Jakarta Commons project (http://jakarta.apache.org/commons/validator/) that can be used with or without Struts. Although Validator is an independent framework again, Struts still comes packaged with it and it is seamlessly integrated.

The Validator framework comes prepackaged with several validation routines, making the transition from hard-coded validation logic painless. Instead of coding validation logic in each Form Bean's **validate( )** method, with Validator you use an XML configuration file to declare the validations that should be applied to each Form Bean. If you need a validation not provided by Validator, you can plug your own custom validations into Validator. Additionally, Validator supports both server-side and client-side (JavaScript) validations whereas Form Beans only provide a server-side validation interface.

Validator Overview

Before getting into the details of using the Validator framework, it's necessary to give an overview of how Validator works. Recall that without Validator, you have to code all of your form data validations into the **validate( )** methods of your Form Bean objects. Each Form Bean field that you want to perform a validation on requires you to code logic to do so. Additionally, you have to write code that will store error messages for validations that fail. With Validator, you don't have to write any code in your Form Beans for validations or storing error messages. Instead, your Form Beans extend one of Validator's **ActionForm** subclasses that provide this functionality for you.

The Validator framework is set up as a pluggable system of validation routines that can be applied to Form Beans. Each validation routine is simply a Java method that is responsible for performing a specific type of validation and can either pass or fail. By default, Validator comes packaged with several useful validation routines that will satisfy most validation scenarios. However, if you need a validation that is not provided by the framework, you can create your own custom validation routine and plug it into the framework.

Validator uses two XML configuration files to tell it which validation routines should be "installed" and how they should be applied for a given application, respectively. The first configuration file, **validator-rules.xml**, declares the validation routines that are plugged into the framework and assigns logical names to each of the validations. Additionally, the **validator-rules.xml** file is used to define client-side JavaScript code (or the location of client-side JavaScript code) for each validation routine. If configured to do so, Validator will emit this JavaScript code to the browser so that validations are performed on the client side as well as the server side. The second configuration file, **validation.xml**, defines which validation routines are applied to which Form Beans. The definitions in this file use the logical names of Form Beans from the Struts configuration file (e.g., **struts-config.xml**) along with the logical names of validation routines from the **validator-rules.xml** file to tie the two together.

Note It's important to point out that while traditionally the **validator-rules.xml** file stores validation routine definitions and the **validation.xml** file applies validation routines to Form Beans, both configuration files are governed by the same DTD. Thus their contents could technically be combined into one file.

Using Validator

Using the Validator framework involves enabling the Validator plugin, configuring Validator's two configuration files, and creating Form Beans that extend Validator's **ActionForm** subclasses. The following sections explain how to configure and use the Validator in detail.

Note Detailed information on configuring the Validator XML configuration files is found in Chapter 20.

Enabling the Validator Plugin

Although the Validator framework comes packaged with Struts, by default Validator is not enabled. In order to enable and use Validator, you have to add to your application's Struts configuration file the following definition for the **plug-in** tag:

```
<!-- Validator Configuration -->
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property property="pathnames"
                   value="/org/apache/struts/validator/validator-
rules.xml,
                          /WEB-INF/validation.xml"/>
</plug-in>
```

This definition causes Struts to load and initialize the Validator plugin for your application. Upon initialization, the plugin loads the comma-delimited list of Validator configuration files specified by the **pathnames** property. Each configuration file's path must be specified by using a Web application–relative path or by using a path to a file on the classpath of the server, as shown in the preceding example. The **validator-rules.xml** file shown in the preceding example happens to be stored in the core Struts **.jar** file, and thus it is accessible via the classpath.

Note that your application's Struts configuration file must conform to the Struts Configuration DTD, which specifies the order in which elements are to appear in the file. Because of this, you must place the Validator **plug-in** tag definition in the proper place in the file. The easiest way to ensure that you are properly ordering elements in the file is to use a tool like Struts Console that automatically formats your configuration file so that it conforms to the DTD.

Creating Form Beans

In order to use Validator, your application's Form Beans must subclass one of Validator's **ActionForm** subclasses instead of **ActionForm** itself. Validator's **ActionForm** subclasses provide an implementation for **ActionForm**'s **reset( )** and **validate( )** methods that hook into the Validator framework. Instead of hard-coding validations into the **validate( )** method, as you would normally do, you simply omit the method altogether because Validator provides the validation code for you.

Parallel to the core functionality provided by Struts, Validator gives you two options to choose from when creating Form Beans. The first option is to create a concrete Form Bean object like the one shown here:

```
package com.jamesholmes.minihr;


import org.apache.struts.validator.ValidatorForm;


public class LogonForm extends ValidatorForm {
  private String username;
  private String password;


  public String getUsername() {
```

```
    return username;
  }

  public void setUsername(String username) {
    this.username = username;
  }

  public String getPassword() {
    return password;
  }

  public void setPassword(String password) {
    this.password = password;
  }
}
```

This class is similar to one that you would create if you were not using Validator; however, this class extends **ValidatorForm** instead of **ActionForm**. This class also does not provide an implementation for **ActionForm**'s empty **reset( )** and **validate( )** methods, because **ValidatorForm** does.

You configure this Form Bean in the Struts configuration file the same way you would a regular Form Bean, as shown here:

```
<form-beans>
  <form-bean name="logonForm"
             type="com.jamesholmes.minihr.LogonForm"/>
</form-beans>
```

The logical name given to the Form Bean with the **form-bean** tag's **name** attribute is the name that you will use when defining validations in the**validation.xml** file, as shown here:

```
<!DOCTYPE form-validation PUBLIC
          "-//Apache Software Foundation//DTD Commons
           Validator Rules Configuration 1.3.0//EN"
          "http://jakarta.apache.org/commons/dtds/validator_1_3_0.dtd">

<form-validation>
  <formset>
    <form name="logonForm">
      <field property="username" depends="required">
        <arg position="0" key="prompt.username"/>
      </field>
    </form>
  </formset>
```

```
</form-validation>
```

Validator uses the value of the **form** tag's **name** attribute to match validation definitions to the name of the Form Bean to which they are applied.

The second option you can choose when creating your Form Bean is to define a Dynamic Form Bean in the Struts configuration file, as shown here:

```
<form-beans>
  <form-bean name="logonForm"
             type="org.apache.struts.validator.DynaValidatorForm">
    <form-property name="username" type="java.lang.String"/>
    <form-property name="password" type="java.lang.String"/>
  </form-bean>
</form-beans>
```

Dynamic Form Beans do not require you to create concrete Form Bean objects; instead, you define the properties that your Form Bean will have and their types, and Struts will dynamically create the Form Bean for you. Validator allows you to use this concept just as you would with core Struts. The only difference for Validator is that you specify that your Form Bean is of type **org.apache.struts.validator.DynaValidatorForm** instead of**org.apache.struts.action.DynaActionForm**.

Identical to the way concrete Form Beans work with Validator, the logical name given to Dynamic Form Beans is the name that you will use when defining validations in the **validation.xml** file. Validator uses the matching names to tie the validations to the Form Bean.

In addition to the two standard options for creating Form Beans, Validator provides an advanced feature for tying multiple validation definitions to one Form Bean definition. When using **ValidatorForm**- or **DynaValidatorForm**-based Form Beans, Validator uses the logical name for the Form Bean from the Struts configuration file to map the Form Bean to validation definitions in the **validation.xml** file. This mechanism is ideal for most cases; however, there are scenarios where Form Beans are shared among multiple actions. One action may use all the Form Bean's fields, and another action may use only a subset of the fields. Because validation definitions are tied to the Form Bean, the action that uses only a subset of the fields has no way of bypassing validations for the unused fields. When the Form Bean is validated, it will generate error messages for the unused fields because Validator has no way of knowing not to validate the unused fields; it simply sees them as missing or invalid.

To solve this problem, Validator provides two additional **ActionForm** subclasses that allow you to tie validations to actions instead of Form Beans. That way you can specify which validations to apply to the Form Bean based on which action is using the Form Bean. For concrete Form Beans, you subclass**org.apache.struts.validator.ValidatorActionForm**, as shown here:

```
public class AddressForm extends ValidatorActionForm {
  ...
}
```

For Dynamic Form Beans, you specify a type of **org.apache.struts.validator.DynaValidatorActionForm** for your Form Bean definition in the Struts configuration file:

```
<form-bean name="addressForm"
```

```
            type="org.apache.struts.validator.DynaValidatorActionForm">
  ...
</form-bean>
```

Inside your **validation.xml** file, you map a set of validations to an action path instead of to a Form Bean name. Here's why: if you have two actions defined, such as Create Address and Edit Address, which use the same Form Bean, then each will have a unique action path. This situation is shown here:

```
<action-mappings>
  <action path="/createAddress"
          type="com.jamesholmes.minihr.CreateAddressAction"
          name="addressForm"/>
  <action path="/editAddress"
          type="com.jamesholmes.minihr.EditAddressAction"
          name="addressForm"/>
</action-mappings>
```

The following **validation.xml** file snippet shows two sets of validations that are intended for the same Form Bean but are distinguished by different action paths:

```
<formset>
  <form name="/createAddress">
    <field property="city" depends="required">
      <arg position="0" key="prompt.city"/>
    </field>
  </form>
  <form name="/editAddress">
    <field property="state" depends="required">
      <arg position="0" key="prompt.state"/>
    </field>
  </form>
</formset>
```

Because your Form Bean subclasses either **ValidatorActionForm** or **DynaValidatorActionForm**, Validator knows to use an action path to find validations instead of the Form Bean's logical name.

Using Validator in Conjunction with the Form Bean's reset( ) and validate( ) Methods

As described, Validator's **ActionForm** subclasses take care of the processing that is normally placed in the **reset( )** and **validate( )** methods of a Form Bean. Thus, you typically do not have an implementation of these two methods in your Form Beans when using Validator. However, it is occasionally necessary to augment the automatic processing provided by Validator with your own custom validations or data resets. To do this, you must override the**reset( )** and/or **validate( )** methods provided by the Validator Form Bean classes and call **super.reset( )** and **super.validate( )** respectively before adding any custom processing to the methods. An example of this is shown next:

```
public void reset(ActionMapping mapping,
```

```
                    HttpServletRequest request) {
  super.reset(mapping, request);

  // Custom reset code goes here.
}

public ActionErrors validate(ActionMapping mapping,
                             HttpServletRequest request) {
  ActionErrors errors = super.validate(mapping, request);
  if (errors == null) {
    errors = new ActionErrors();
  }

  // Custom validation code goes here.

  return errors;
}
```

Notice that the call to **super.validate( )** in the **validate( )** method returns an **ActionErrors** instance. Errors generated from custom validations should be added to that **ActionErrors** instance and that instance should be used as the method's return value.

Configuring validator-rules.xml

The Validator framework is set up as a pluggable system whereby each of its validation routines is simply a Java method that is plugged into the system to perform a specific validation. The **validator-rules.xml** file is used to declaratively plug in the validation routines that Validator will use for performing validations. Struts comes packaged with a preconfigured copy of this file in the Struts core **.jar** file (e.g., **struts-core-1.3.5.jar**). Under most circumstances, you will use this preconfigured copy and will not ever need to modify it. Modification to the file would require extracting it from the core **.jar**file, making changes to the file and then repackaging the core **.jar** file with the modified file. As you can imagine, that is cumbersome and should only be done if absolutely necessary. Otherwise you can simply add validation routine definitions to the **validation.xml** file as explained in the section "Creating Custom Validations."

Following is a sample **validator-rules.xml** file that illustrates how validation routines are plugged into Validator:

```
<!DOCTYPE form-validation PUBLIC
          "-//Apache Software Foundation//DTD Commons
           Validator Rules Configuration 1.3.0//EN"
          "http://jakarta.apache.org/commons/dtds/validator_1_3_0.dtd">



<form-validation>
```

```
  <global>
    <validator name="minlength"
          classname="org.apache.struts.validator.FieldChecks"
             method="validateMinLength"
       methodParams="java.lang.Object,
                     org.apache.commons.validator.ValidatorAction,
                     org.apache.commons.validator.Field,
                     org.apache.struts.action.ActionMessages,
                     org.apache.commons.validator.Validator,
                     javax.servlet.http.HttpServletRequest"
                msg="errors.minlength"

jsFunction="org.apache.commons.validator.javascript.validateMinLength"/>
  </global>
</form-validation>
```

Each validation routine in the **validator-rules.xml** file has its own definition that is declared with a **validator** tag. The **validator** tag is used to assign a logical name to the routine, with the **name** attribute, and to specify the class and method for the routine. The logical name given to the routine will be used to refer to the routine by other routines in this file as well as by validation definitions in the **validation.xml** file.

Notice that the **validator** tag specifies a **msg** attribute. The **msg** attribute specifies a key for a message in the application resource bundle file that will be used as the error message when the validation fails. Notice also that the **validator** tag specifies a **jsFunction** attribute. The **jsFunction** attribute is used to define the path to a file that contains client-side JavaScript code for the validation routine. The JavaScript code performs the same validation on the client side as is performed on the server side.

Note In lieu of using the **jsFunction** attribute of the **validator** tag to declare the file in which the routine's client-side JavaScript code is housed, you can nest a **javascript** tag beneath the **validator** tag to specify the JavaScript code for a validation routine.

Configuring the Application Resource Bundle File

Validator uses the Struts Resource Bundle mechanism for externalizing error messages. Instead of having hard-coded error messages in the framework, Validator allows you to specify a key to a message in the application resource bundle file (e.g., **MessageResources. properties**) that is returned if a validation fails. Each validation routine in the **validator-rules. xml** file specifies an error message key with the **validator** tag's **msg** attribute, as shown here:

```
<validator name="minlength"
      classname="org.apache.struts.validator.FieldChecks"
         method="validateMinLength"
   methodParams="java.lang.Object,
                 org.apache.commons.validator.ValidatorAction,
                 org.apache.commons.validator.Field,
```

```
                org.apache.struts.action.ActionMessages,
                org.apache.commons.validator.Validator,
                javax.servlet.http.HttpServletRequest"
          msg="errors.minlength"

jsfunction="org.apache.commons.validator.javascript.validateMinLength"/>
```

If the validation fails when it is run, the message corresponding to the key specified by the **msg** attribute will be returned.

The following snippet shows the default set of validation error messages from the **MessageResources.properties** file that comes prepackaged with the Struts example applications. Each message key corresponds to those specified by the validation routines in the **validator-rules.xml** file that also comes prepackaged with the Struts example applications:

```
# Error messages for Validator framework validations
errors.required={0} is required.
errors.minlength={0} cannot be less than {1} characters.
errors.maxlength={0} cannot be greater than {1} characters.
errors.invalid={0} is invalid.
errors.byte={0} must be a byte.
errors.short={0} must be a short.
errors.integer={0} must be an integer.
errors.long={0} must be a long.
errors.float={0} must be a float.
errors.double={0} must be a double.
errors.date={0} is not a date.
errors.range={0} is not in the range {1} through {2}.
errors.creditcard={0} is not a valid credit card number.
errors.email={0} is an invalid e-mail address.
errors.url={0} is an invalid URL.
```

Notice that each message has placeholders in the form of {0}, {1}, or {2}. At run time, the placeholders will be substituted for another value, such as the name of the field being validated. This feature is known as *parametric replacement* and is especially useful in allowing you to create generic validation error messages that can be reused for several different fields of the same type.

Take for example the **required** validation's error message, **errors.required**:

```
errors.required={0} is required.
```

When you use the **required** validation in the **validation.xml** file, you have to define the value that should be used to substitute {0} in the error message:

```
<form name="auctionForm">
  <field property="bid" depends="required">
    <arg position="0" key="prompt.bid"/>
```

```
  </field>
</form>
```

Error messages can have multiple placeholders: {0} – {N}. These placeholders are specified using the **arg** tag. In the preceding example, the **arg** tag specifies the value that will replace the {0} placeholder. This tag's **key** attribute specifies a message key from the application resource bundle file, such as the one shown next, whose value will be used as the replacement for the placeholder:

```
prompt.bid=Auction Bid
```

Using a message key for the placeholder value frees you from having to hard-code the replacement value over and over in the **validation.xml** file. However, if you don't want to use the Resource Bundle key/value mechanism to specify placeholder values, you can explicitly specify the placeholder value by using the following syntax for the **arg** tag:

```
<arg position="0" key="Auction Bid" resource="false"/>
```

In this example, the **resource** attribute is set to *false*, instructing Validator that the value specified with the **key** attribute should be taken as the literal placeholder value and not as a key for a message in the application resource bundle file.

Configuring validation.xml

The **validation.xml** file is used to declare sets of validations that should be applied to Form Beans. Each Form Bean that you want to validate has its own definition in this file. Inside that definition specify the validations that you want to apply to the Form Bean's fields. Following is a sample**validation.xml** file that illustrates how validations are defined:

```
<!DOCTYPE form-validation PUBLIC
          "-//Apache Software Foundation//DTD Commons
           Validator Rules Configuration 1.3.0//EN"
          "http://jakarta.apache.org/commons/dtds/validator_1_3_0.dtd">


<form-validation>
  <formset>
    <form name="logonForm">
      <field property="username" depends="required">
        <arg position="0" key="prompt.username"/>
      </field>
      <field property="password" depends="required">
        <arg position="0" key="prompt.password"/>
      </field>
    </form>
  </formset>
</form-validation>
```

The first element in the **validation.xml** file is the **<form-validation>** element. This element is the master element for the file and is defined only once. Inside the **<form-validation>** element you define **<form-set>** elements that encapsulate multiple **<form>** elements. Generally, you will define

only one**<form-set>** element in your file; however, you would use a separate one for each locale if you were internationalizing validations. Internationalizing validations is covered later in this chapter.

Each **<form>** element uses the **name** attribute to associate a name with the set of field validations it encompasses. Validator uses this logical name to map the validations to a Form Bean defined in the Struts configuration file. Based on the type of Form Bean being validated, Validator will attempt to match the name either against a Form Bean's logical name or against an action's path. Inside the **<form>** element, **<field>** elements are used to define the validations that will be applied to specified Form Bean fields. The **<field>** element's **property** attribute corresponds to the name of a field in the specified Form Bean. The **depends** attribute specifies the logical names of validation routines from the **validatorrules.xml** file that should be applied to the field. The validations specified with the **depends** attribute will be performed in the order specified and they all must pass.

Working with Configurable Validations

Each of the prepackaged validations provided by Validator requires configuring before it will function properly. There are two types of configuration constructs that validations can use: error message parametric replacement definitions and variable definitions. Error message parametric replacement definitions are defined with the **arg** tag and specify values for placeholders in the error message associated with a validation. Before an error message is generated for a failed validation, the message's placeholders are replaced with the values specified with **arg** tags. Variable definitions are defined with the **var** tag and specify validation-specific configuration settings. Validations use the configuration settings specified with **var** tags to guide their behavior.

Following is an example of how to use the **arg** tag to specify a parametric replacement value for a validation's error message:

```
<field property="password" depends="required, minlength">
  <arg position="0" key="prompt.password"/>
</field>
```

The **position** attribute of the **arg** tag is used to specify which parameter will be replaced with the given value. For example, position "0" is used to replace "{0}" in a message, position "1" is used to replace "{1}" and so on. An example error message with a replacement placeholder is shown here:

```
errors.required={0} is required.
```

The **key** attribute of the **arg** tag specifies a key for a value in the application resource bundle file that will be used to populate the validation's error message. Alternatively, you can set the **resource** attribute of the **arg** tag to "false" to indicate that the value specified with the **key** attribute should be taken as the literal replacement value instead of as the key for a value in the resource bundle. An example of specifying a literal replacement value is shown next:

```
<field property="password" depends="required, minlength">
  <arg position="0" key="Password" resource="false"/>
</field>
```

The value "Password" will be used to replace the "{0}" in the validations' error messages.

Each validation applied to a field via the **depends** attribute of the **field** tag will use the same parametric replacement values specified with **arg** tags unless an **arg** tag explicitly specifies the validation it is to be used for with the **name** attribute, as shown next:

```
<field property="password" depends="required, minlength">
  <arg position="0" name="required" key="prompt.passwordReq"/>
```

```
  <arg position="0" name="minlength" key="prompt.passwordMin"/>
</field>
```

The **name** attribute specifies the name of a specific validation that the parametric replacement value is for.

Next is an example of how to use the **var** tag to specify a configuration setting for a validation:

```
<field property="socialSecurityNum" depends="required, mask">
  <arg position="0" key="label.sampleForm.socialSecurityNum"/>
  <var>
    <var-name>mask</var-name>
    <var-value>^\d{3}-\d{2}-\d{4}$</var-value>
  </var>
</field>
```

Inside the **var** tag you must nest **var-name** and **var-value** tags to specify the name and value of a configuration setting, respectively. The names of acceptable configuration settings specified with the **var-name** tag are validation-specific and can be found in the sections dedicated to each validation later in this chapter. With some validations it is useful to use the value set for a variable with the **var-value** tag as the value for a parametric replacement in the error message. For example, the **minlength** validation's error message has a placeholder for minimum length. Instead of specifying the minimum length twice, once for the error message and once for the validation configuration setting, you can use the value of the configuration setting as the value for the placeholder. The example shown next illustrates how to do this:

```
<field property="password" depends="required, minlength">
  <arg position="0" key="label.sampleForm.password"/>
  <arg position="1" key="${var:minlength}" resource="false"/>
  <var>
    <var-name>minlength</var-name>
    <var-value>4</var-value>
  </var>
</field>
```

Notice that the second **arg** tag's **key** attribute is set to "${var:minlength}" and the **resource** attribute is set to "false". Recall that setting the **resource**attribute to false indicates that the value specified with the **key** attribute is to be taken as a literal value instead of as a key from the application resource bundle file. The "${var:minlength}" value of the **key** attribute indicates that the literal value should be the value of the **minlength** variable defined with the**var** tag.

Extending a Set of Validations

As of Commons Validator version 1.2, Validator has a feature for extending form definitions. This powerful mechanism is analogous to inheritance in Java and makes setting up similar forms very simple. For example, if you have a registration form with several fields and you need to define another form that has all of the same fields as the registration form plus a few additional fields, you can simply extend the registration form and add only the additional fields to the second form's definition. Following is an example of how this is done:

```
<form-validation>
```

```
  <formset>
    <form name="registrationForm">
      <field property="name" depends="required">
        <arg position="0" key="prompt.name"/>
      </field>
      <field property="address" depends="required">
        <arg position="0" key="prompt.address"/>
      </field>
    </form>
    <form name="businessRegistrationForm" extends="registrationForm">
      <field property="suite" depends="required">
        <arg position="0" key="prompt.suite"/>
      </field>
    </form>
  </formset>
</form-validation>
```

Notice in the example that the form definition for the **businessRegistrationForm** uses the **form** tag's **extends** attribute to specify another form that the form extends. All of the validation configuration details from the **registrationForm** form definition will be added to the **businessRegistrationForm** form defintion. Again, as with Java's inheritance feature, you can have any number of levels of extension.

Creating Validations for Indexed Properties

As mentioned in Chapter 4, indexed properties are a powerful and convenient mechanism for working with lists of objects on form beans. Validator provides built-in support for validating indexed properties via the **indexedListProperty** attribute of the **field** tag, as shown here:

```
<form name="registrationForm">
  <field property="firstName"
      indexedListProperty="customer" depends="required">
    <arg position="0" key="prompt.firstName"/>
  </field>
  <field property="lastName"
      indexedListProperty="customer" depends="required">
    <arg position="0" key="prompt.lastName"/>
  </field>
</form>
```

In this example, the **registrationForm** has a **customer** indexed property, as specified with the **indexedListProperty** attribute of the **field** tag. Validator will loop over each object in the **customer** indexed property and validate the field specified with the **property** attribute. This scenario assumes that the property specified with the **indexedListProperty** attribute is a collection of objects and the field specified with the **property** attribute has getter and setter methods on the individual objects in the collection.

Validations That Span Multiple Pages

Web applications often use wizard-like page flows to break up large monolithic forms into multiple pages to reduce the amount of information that must be entered on a single page. In that scenario, it is commonplace to use one Form Bean to store the data collected across all of the pages. At the end of the wizard page flow an action will process all of the collected data at once. The problem with that scenario, validation-wise, is that validations are specified per form in the **validation.xml** file and not all validations should be run against the form on each page. For example, the fields on page 2 shouldn't be validated on page 1 because they have not been entered yet. Any validations for page 2 that are run before page 2 is reached will fail. This is the case for each successive page in the wizard page flow. Validator has a feature to handle this situation so that validations are run only at the appropriate times.

In order to limit when validations are run for a form that spans multiple pages, the **field** tag has a **page** attribute that is used to specify a page number, as shown next:

```
<form name="registrationForm">
  <field property="name" page="1" depends="required">
    <arg position="0" key="prompt.name"/>
  </field>
  <field property="address" page="1" depends="required">
    <arg position="0" key="prompt.address"/>
  </field>
  <field property="username" page="2" depends="required">
    <arg position="0" key="prompt.username"/>
  </field>
  <field property="password" page="2" depends="required">
    <arg position="0" key="prompt.password"/>
  </field>
</form>
```

The page number specified with the **field** tag's **page** attribute corresponds to the value of a field on the associated form bean that must be set in the JSP page:

```
<html:hidden property="page" value="1"/>
```

The page property is already on the Validator **ActionForm** subclasses, so you don't need to add it to your Form Beans.

Each page must use the HTML Tag Library's **hidden** tag to specify a unique value for the **page** property. When validations are run, Validator compares the page value from the form with the page value set on the **field** tag with the **page** attribute. If the values are less than or equal to one another, the validations for the given field are run. Otherwise, the field's validations are skipped. Note that if you need to apply any custom reset logic on your Form Bean's fields using the **reset( )** method, you can use the **page** property, as shown here:

```
public void reset(ActionMapping mapping,
                  HttpServletRequest request) {
  super.reset(mapping, request);
```

```
  // Custom reset code goes here.
  if (page == 1) {
    // Reset page 1 properties.
  }
  else if (page == 2) {
    // Reset page 2 properties.
  }
}
```

As mentioned, the **page** property is a protected member of the Validator **ActionForm** subclasses, and thus it is available to Form Beans that extend those subclasses.

Using Validator's Included Validations

Validator, by default, includes several basic validation routines that you can use to solve most validation scenarios. As mentioned, Struts comes packaged with a preconfigured **validator-rules.xml** file that defines these routines. Table 6-1 lists each of the preconfigured validations by logical name and states its purpose. The following sections describe each of the preconfigured validations listed in the table, and usage examples are given.

Table 6-1: Validator's Preconfi gured Validations

Name	Description
byte	Determines whether the field being validated contains a value that can be converted to a Java **byte** primitive type.
byteLocale	Determines whether the field being validated contains a value that can be converted to a Java **byte** primitive type using the number formatting conventions of the current user's locale.
creditCard	Determines whether the field being validated contains a valid credit card number from one of the four major credit card companies (American Express, Discover, MasterCard, or Visa).
date	Determines whether the field being validated contains a value that can be converted to a **java.util.Date** type using the **java.text.SimpleDateFormat** class.
double	Determines whether the field being validated contains a value that can be converted to a Java **double**primitive type.
doubleRange	Determines whether the field being validated contains a Java **double** primitive type value that falls within the specified range.
email	Determines whether the field being validated contains a value that is a properly formatted e-mail address.
float	Determines whether the field being validated contains a value that can be converted to a Java **float** primitive type.
floatLocale	Determines whether the field being validated contains a value that can be converted to a Java **float** primitive type using the number formatting conventions of the current user's locale.
floatRange	Determines whether the field being validated contains a Java **float** primitive type value that falls within the specified range.

Table 6-1: Validator's Preconfi gured Validations

Name	Description
integer	Determines whether the field being validated contains a value that can be converted to a Java **int** primitive type.
integerLocale	Determines whether the field being validated contains a value that can be converted to a Java **int** primitive type using the number formatting conventions of the current user's locale.
intRange	Determines whether the field being validated contains a Java **int** primitive type value that falls within the specified range.
long	Determines whether the field being validated contains a value that can be converted to a Java **long** primitive type.
longLocale	Determines whether the field being validated contains a value that can be converted to a Java **long** primitive type using the number formatting conventions of the current user's locale.
longRange	Determines whether the field being validated contains a Java **long** primitive type value that falls within the specified range.
mask	Determines whether the field being validated contains a value that is properly formatted according to a specified regular expression.
maxlength	Determines whether the field being validated contains a value whose character length is less than the specified maximum length.
minlength	Determines whether the field being validated contains a value whose character length is more than the specified minimum length.
required	Determines whether the field being validated contains a value other than white space (i.e., space, tab, and newline characters).
requiredif	Deprecated. Originally used for creating conditional validations. Use the **validwhen** validation instead.
short	Determines whether the field being validated contains a value that can be converted to a Java **short** primitive type.
shortLocale	Determines whether the field being validated contains a value that can be converted to a Java **short** primitive type using the number formatting conventions of the current user's locale.
url	Determines whether the field being validated contains a value that is a properly formatted URL.
validwhen	Determines whether the field being validated is required based on a specified test condition.

The byte Validation

The **byte** validation is used to determine whether the field being validated contains a value that can be converted to a Java **byte** primitive type. If the conversion succeeds, the validation passes. The following snippet illustrates how to configure a **byte** validation in a Validator configuration file:

```
<field property="testByte" depends="required, byte">
  <arg position="0" key="label.sampleForm.testByte"/>
</field>
```

Configuring the **byte** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the **byte**validation's error message.

The **byte** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.byte** entry in the resource bundle file for its error message. The default **errors.byte** entry is shown next:

```
errors.byte={0} must be a byte.
```

The byteLocale Validation

The **byteLocale** validation is used to determine whether the field being validated contains a value that can be converted to a Java **byte** primitive type using the number formatting conventions of the current user's locale. If the conversion succeeds, the validation passes. This validation works identically to the **byte** validation except that it uses the user's locale to constrain the values that will be valid. The following snippet illustrates how to configure a**byteLocale** validation in a Validator configuration file:

```
<field property="testByte" depends="required, byteLocale">
  <arg position="0" key="label.sampleForm.testByte"/>
</field>
```

Configuring the **byteLocale** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field**tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the**byteLocale** validation's error message.

The **byteLocale** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.byte** entry in the resource bundle file for its error message. The default **errors. byte** entry is shown next:

```
errors.byte={0} must be a byte.
```

The creditCard Validation

The **creditCard** validation is used to determine whether the field being validated contains a valid credit card number from one of the four major credit card companies (American Express, Discover, MasterCard, or Visa). If the credit card number passes a checksum routine, the validation passes. The following snippet illustrates how to configure a **creditCard** validation in a Validator configuration file:

```
<field property="cardNumber" depends="required, creditCard">
  <arg position="0" key="label.sampleForm.cardNumber"/>
</field>
```

Configuring the **creditCard** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field**tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the**creditCard** validation's error message.

The **creditCard** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.creditcard** entry in the resource bundle file for its error message. The default **errors.creditcard** entry is shown next:

```
errors.creditcard={0} is an invalid credit card number.
```

The date Validation

The **date** validation is used to determine whether the field being validated contains a value that can be converted to a **java.util.Date** type using the**java.text.SimpleDateFormat** class. If the conversion succeeds, the validation passes. The following snippet illustrates how to configure a **date**validation in a Validator configuration file:

```
<field property="birthDate" depends="required, date">
  <arg position="0" key="label.sampleForm.birthDate"/>
  <var>
    <var-name>datePattern</var-name>
    <var-value>MM/dd/yyyy</var-value>
  </var>
</field>
```

or

```
<field property="birthDate" depends="required, date">
  <arg position="0" key="label.sampleForm.birthDate"/>
  <var>
    <var-name>datePatternStrict</var-name>
    <var-value>MM/dd/yyyy</var-value>
  </var>
</field>
```

To configure the **date** validation, you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must use an **arg**tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the **date** validation's error message. Additionally, you must use the **var** tag to define a **datePattern** or **datePatternStrict** variable. The **datePattern** variable is used by the **date** validation to specify the pattern that dates must have in order to pass the validation.
The **datePatternStrict** variable works the same way as the **datePattern** variable except that it requires that dates adhere strictly to the pattern by matching the pattern's length. For example, a date value of "6/12/2006" would not match the pattern of "MM/dd/yyyy" with the **datePatternStrict** variable because it wasn't "06/12/2006" (notice the leading zero on the day). The abbreviated date would, however, match with the **datePattern** variable. The **datePattern** and **datePatternStrict** variables accept any pattern accepted by the**java.text.SimpleDateFormat** class.

The **date** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.date** entry in the resource bundle file for its error message. The default **errors.date** entry is shown next:

```
errors.date={0} is not a date.
```

The double Validation

The **double** validation is used to determine whether the field being validated contains a value that can be converted to a Java **double** primitive type. If the conversion succeeds, the validation passes. The following snippet illustrates how to configure a **double** validation in a Validator configuration file:

```
<field property="orderTotal" depends="required, double">
  <arg position="0" key="label.sampleForm.orderTotal"/>
</field>
```

Configuring the **double** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the **double**validation's error message.

The **double** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.double** entry in the resource bundle file for its error message. The default **errors.double** entry is shown next:

```
errors.double={0} must be a double.
```

The doubleRange Validation

The **doubleRange** validation is used to determine whether the field being validated contains a Java **double** primitive type value that falls within the specified range. If the value is within the specified range, the validation passes. The following snippet illustrates how to configure a **doubleRange**validation in a Validator configuration file:

```
<field property="orderTotal" depends="required, doubleRange">
  <arg position="0" key="label.sampleForm.orderTotal"/>
  <arg position="1" key="${var:min}" resource="false"/>
  <arg position="2" key="${var:max}" resource="false"/>
  <var>
    <var-name>min</var-name>
    <var-value>100</var-value>
  </var>
  <var>
    <var-name>max</var-name>
    <var-value>1000</var-value>
  </var>
</field>
```

To configure the **doubleRange** validation, you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must use **arg** tags to specify the keys for values in the application resource bundle file or literal values that will be used to populate the **doubleRange**validation's error message. Additionally, you must use **var** tags to define **min** and **max** variables. The **min** and **max** variables are used by the**doubleRange** validation to specify the minimum and maximum values for the range, respectively. Notice that the **arg** tags' keys are set to **$var:min** and**$var:max**. The **min** and **max** values specified with the **var** tags will be used to populate the error message, respectively.

The **doubleRange** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.range** entry in the resource bundle file for its error message. The default **errors.range** entry is shown next:

```
errors.range={0} is not in the range {1} through {2}.
```

The email Validation

The **email** validation is used to determine whether the field being validated contains a value that is a properly formatted e-mail address. If the format is correct, the validation passes. The following snippet illustrates how to configure an **email** validation in a Validator configuration file:

```
<field property="emailAddress" depends="required, email">
  <arg position="0" key="label.sampleForm.emailAddress"/>
</field>
```

Configuring the **email** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the **email**validation's error message.

The **email** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.email** entry in the resource bundle file for its error message. The default **errors.email** entry is shown next:

```
errors.email={0} is an invalid e-mail address.
```

The float Validation

The **float** validation is used to determine whether the field being validated contains a value that can be converted to a Java **float** primitive type. If the conversion succeeds, the validation passes. The following snippet illustrates how to configure a **float** validation in a Validator configuration file:

```
<field property="orderTotal" depends="required, float">
  <arg position="0" key="label.sampleForm.orderTotal"/>
</field>
```

Configuring the **float** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the **float**validation's error message.

The **float** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.float** entry in the resource bundle file for its error message. The default **errors.float** entry is shown next:

```
errors.float={0} must be a float.
```

The floatLocale Validation

The **floatLocale** validation is used to determine whether the field being validated contains a value that can be converted to a Java **float** primitive type using the number formatting conventions of the current user's locale. If the conversion succeeds, the validation passes. This validation works identically to the **float** validation except that it uses the user's locale to constrain the values that will be valid. The following snippet illustrates how to configure a**floatLocale** validation in a Validator configuration file:

```
<field property="orderTotal" depends="required, floatLocale">
  <arg position="0" key="label.sampleForm.orderTotal"/>
</field>
```

Configuring the **floatLocale** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field**tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the**floatLocale** validation's error message.

The **floatLocale** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.float** entry in the resource bundle file for its error message. The default **errors. float** entry is shown next:

```
errors.float={0} must be a float.
```

The floatRange Validation

The **floatRange** validation is used to determine whether the field being validated contains a Java **float** primitive type value that falls within the specified range. If the value is within the range, the validation passes. The following snippet illustrates how to configure a **floatRange** validation in a Validator configuration file:

```
<field property="orderTotal" depends="required, floatRange">
  <arg position="0" key="label.sampleForm.orderTotal"/>
  <arg position="1" key="${var:min}" resource="false"/>
  <arg position="2" key="${var:max}" resource="false"/>
  <var>
    <var-name>min</var-name>
    <var-value>100</var-value>
  </var>
  <var>
    <var-name>max</var-name>
    <var-value>1000</var-value>
  </var>
</field>
```

To configure the **floatRange** validation, you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must use**arg** tags to specify the keys for values in the application resource bundle file or literal values that will be used to populate the **floatRange** validation's error message. Additionally, you must use **var** tags to define **min** and **max** variables. The **min** and **max** variables are used by the **floatRange** validation to specify the minimum and maximum values for the range, respectively. Notice that the **arg** tags' keys are set to **$var:min** and **$var:max**. The **min** and**max** values specified with the **var** tags will be used to populate the error message, respectively.

The **floatRange** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.range** entry in the resource bundle file for its error message. The default **errors. range** entry is shown next:

```
errors.range={0} is not in the range {1} through {2}.
```

The integer Validation

The **integer** validation is used to determine whether the field being validated contains a value that can be converted to a Java **int** primitive type. If the conversion succeeds, the validation passes. The following snippet illustrates how to configure an **integer** validation in a Validator configuration file:

```
<field property="productCount" depends="required, integer">
  <arg position="0" key="label.sampleForm.productCount"/>
</field>
```

Configuring the **integer** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the **integer**validation's error message.

The **integer** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.integer** entry in the resource bundle file for its error message. The default **errors.integer** entry is shown next:

```
errors.integer={0} must be an integer.
```

The integerLocale Validation

The **integerLocale** validation is used to determine whether the field being validated contains a value that can be converted to a Java **int** primitive type using the number formatting conventions of the current user's locale. If the conversion succeeds, the validation passes. This validation works identically to the **integer** validation except that it uses the user's locale to constrain the values that will be valid. The following snippet illustrates how to configure an**integerLocale** validation in a Validator configuration file:

```
<field property="productCount" depends="required, integerLocale">
  <arg position="0" key="label.sampleForm.productCount"/>
</field>
```

Configuring the **integerLocale** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field**tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the**integerLocale** validation's error message.

The **integerLocale** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.integer** entry in the resource bundle file for its error message. The default **errors.integer** entry is shown next:

```
errors.integer={0} must be an integer.
```

The intRange Validation

The **intRange** validation is used to determine whether the field being validated contains a Java **int** primitive type value that falls within the specified range. If the value is within the range, the validation passes. The following snippet illustrates how to configure an **intRange** validation in a Validator configuration file:

```
<field property="orderTotal" depends="required, intRange">
  <arg position="0" key="label.sampleForm.orderTotal"/>
  <arg position="1" key="${var:min}" resource="false"/>
  <arg position="2" key="${var:max}" resource="false"/>
  <var>
    <var-name>min</var-name>
    <var-value>100</var-value>
  </var>
  <var>
    <var-name>max</var-name>
    <var-value>1000</var-value>
```

```
  </var>
</field>
```

To configure the **intRange** validation, you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must use**arg** tags to specify the keys for values in the application resource bundle file or literal values that will be used to populate the **intRange** validation's error message. Additionally, you must use **var** tags to define **min** and **max** variables.
The **min** and **max** variables are used by the **intRange** validation to specify the minimum and maximum values for the range, respectively. Notice that the **arg** tags' keys are set to **$var:min** and **$var:max**. The **min** and**max** values specified with the **var** tags will be used to populate the error message, respectively.

The **intRange** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.range** entry in the resource bundle file for its error message. The default **errors.range** entry is shown next:

```
errors.range={0} is not in the range {1} through {2}.
```

The long Validation

The **long** validation is used to determine whether the field being validated contains a value that can be converted to a Java **long** primitive type. If the conversion succeeds, the validation passes. The following snippet illustrates how to configure a **long** validation in a Validator configuration file:

```
<field property="productCount" depends="required, long">
  <arg position="0" key="label.sampleForm.productCount"/>
</field>
```

Configuring the **long** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the **long**validation's error message.

The **long** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.long** entry in the resource bundle file for its error message. The default **errors.long** entry is shown next:

```
errors.long={0} must be a long.
```

The longLocale Validation

The **longLocale** validation is used to determine whether the field being validated contains a value that can be converted to a Java **long** primitive type using the number formatting conventions of the current user's locale. If the conversion succeeds, the validation passes. This validation works identically to the **long** validation except that it uses the user's locale to constrain the values that will be valid. The following snippet illustrates how to configure a**longLocale** validation in a Validator configuration file:

```
<field property="productCount" depends="required, longLocale">
  <arg position="0" key="label.sampleForm.productCount"/>
</field>
```

Configuring the **longLocale** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field**tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the**longLocale** validation's error message.

The **longLocale** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.long** entry in the resource bundle file for its error message. The default **errors. long** entry is shown next:

```
errors.long={0} must be a long.
```

The longRange Validation

The **longRange** validation is used to determine whether the field being validated contains a Java **long** primitive type value that falls within the specified range. If the value is within the range, the validation passes. The following snippet illustrates how to configure a **longRange** validation in a Validator configuration file:

```
<field property="orderTotal" depends="required, longRange">
  <arg position="0" key="label.sampleForm.orderTotal"/>
  <arg position="1" key="${var:min}" resource="false"/>
  <arg position="2" key="${var:max}" resource="false"/>
  <var>
    <var-name>min</var-name>
    <var-value>100</var-value>
  </var>
  <var>
    <var-name>max</var-name>
    <var-value>1000</var-value>
  </var>
</field>
```

To configure the **longRange** validation, you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must use**arg** tags to specify the keys for values in the application resource bundle file or literal values that will be used to populate the **longRange** validation's error message. Additionally, you must use **var** tags to define **min** and **max** variables. The **min** and **max** variables are used by the **longRange** validation to specify the minimum and maximum values for the range, respectively. Notice that the **arg** tags' keys are set to **$var:min** and **$var:max**. The **min** and**max** values specified with the **var** tags will be used to populate the error message, respectively.

The **longRange** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.range** entry in the resource bundle file for its error message. The default **errors. range** entry is shown next:

```
errors.range={0} is not in the range {1} through {2}.
```

The mask Validation

The **mask** validation is used to determine whether the field being validated contains a value that is properly formatted according to a specified regular expression. If the value's format matches the regular expression, the validation passes. The following snippet illustrates how to configure a **mask**validation in a Validator configuration file:

```
<field property="socialSecurityNum" depends="required, mask">
  <arg position="0" key="label.sampleForm.socialSecurityNum"/>
  <var>
```

```
    <var-name>mask</var-name>
    <var-value>^\d{3}-\d{2}-\d{4}$</var-value>
  </var>
</field>
```

To configure the **mask** validation, you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must use an **arg**tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the **mask** validation's error message. Additionally, you must use the **var** tag to define a **mask** variable. The **mask** variable is used by the **mask** validation to specify a regular expression to which values must conform in order to be deemed valid.

Note Regular expressions are a powerful tool for text analysis and manipulation; however, they constitute a large, and at times complex, topic. Therefore, a discussion of regular expressions is outside the scope of this book.

The **mask** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.invalid** entry in the resource bundle file for its error message. The default **errors.invalid** entry is shown next:

```
errors.invalid={0} is invalid.
```

The maxlength Validation

The **maxlength** validation is used to determine whether the field being validated contains a value whose character length is less than the specified maximum length. If the value contains fewer characters than the specified maximum, the validation passes. The following snippet illustrates how to configure a **maxlength** validation in a Validator configuration file:

```
<field property="password" depends="required, maxlength">
  <arg position="0" key="label.sampleForm.password"/>
  <arg position="1" key="${var:maxlength}" resource="false"/>
  <var>
    <var-name>maxlength</var-name>
    <var-value>8</var-value>
  </var>
</field>
```

To configure the **maxlength** validation, you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must use**arg** tags to specify the keys for values in the application resource bundle file or literal values that will be used to populate the **maxlength** validation's error message. Additionally, you must use the **var** tag to define a **maxlength** variable. The **maxlength** variable is used by the **maxlength** validation to specify the maximum character length that values can have. Notice that the **arg** tag's key is set to **$var:maxlength**. The **maxlength** value specified with the **var** tag will be used to populate the error message.

The **maxlength** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.maxlength** entry in the resource bundle file for its error message. The default **errors.maxlength** entry is shown next:

```
errors.maxlength={0} can not be greater than {1} characters.
```

The minlength Validation

The **minlength** validation is used to determine whether the field being validated contains a value whose character length is more than the specified minimum length. If the value contains more characters than the specified minimum, the validation passes. The following snippet illustrates how to configure a **minlength** validation in a Validator configuration file:

```
<field property="password" depends="required, minlength">
  <arg position="0" key="label.sampleForm.password"/>
  <arg position="1" key="${var:minlength}" resource="false"/>
  <var>
    <var-name>minlength</var-name>
    <var-value>4</var-value>
  </var>
</field>
```

To configure the **minlength** validation, you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must use**arg** tags to specify the keys for values in the application resource bundle file or literal values that will be used to populate the **minlength** validation's error message. Additionally, you must use the **var** tag to define a **minlength** variable. The **minlength** variable is used by the **minlength** validation to specify the minimum character length that values can have. Notice that the **arg** tag's key is set to **$var:minlength**. The **minlength** value specified with the **var**tag will be used to populate the error message.

The **minlength** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.minlength** entry in the resource bundle file for its error message. The default **errors.minlength** entry is shown next:

```
errors.minlength={0} can not be less than {1} characters.
```

The required Validation

The **required** validation is used to determine whether the field being validated contains a value other than white space (i.e., space, tab, and newline characters). If a non–white space value is present, the validation passes. The following snippet illustrates how to configure a **required** validation in a Validator configuration file:

```
<field property="name" depends="required">
  <arg position="0" key="label.sampleForm.name"/>
</field>
```

Configuring the **required** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the**required** validation's error message.

The **required** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.required** entry in the resource bundle file for its error message. The default **errors.required** entry is shown next:

```
errors.required={0} is required.
```

The short Validation

The **short** validation is used to determine whether the field being validated contains a value that can be converted to a Java **short** primitive type. If the conversion succeeds, the validation passes. The following snippet illustrates how to configure a **short** validation in a Validator configuration file:

```
<field property="productCount" depends="required, short">
  <arg position="0" key="label.sampleForm.productCount"/>
</field>
```

Configuring the **short** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the **short**validation's error message.

The **short** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.short** entry in the resource bundle file for its error message. The default **errors.short** entry is shown next:

```
errors.short={0} must be a short.
```

The shortLocale Validation

The **shortLocale** validation is used to determine whether the field being validated contains a value that can be converted to a Java **short** primitive type using the number formatting conventions of the current user's locale. If the conversion succeeds, the validation passes. This validation works identically to the **short** validation except that it uses the user's locale to constrain the values that will be valid. The following snippet illustrates how to configure a**shortLocale** validation in a Validator configuration file:

```
<field property="productCount" depends="required, shortLocale">
  <arg position="0" key="label.sampleForm.productCount"/>
</field>
```

Configuring the **shortLocale** validation is straightforward: you specify it in the list of validations for the given field with the **depends** attribute of the **field**tag. You must also use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the**shortLocale** validation's error message.

The **shortLocale** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.short** entry in the resource bundle file for its error message. The default **errors.short** entry is shown next:

```
errors.short={0} must be a short.
```

The url Validation

The **url** validation is used to determine whether the field being validated contains a value that is a properly formatted URL. If the format is correct, the validation passes. The following snippet illustrates how to configure a **url** validation in a Validator configuration file:

```
<field property="websiteUrl" depends="required, url">
  <arg position="0" key="label.sampleForm.websiteUrl"/>
</field>
```

To configure the **url** validation, you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must use an **arg**tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the **url** validation's error

message. Additionally, you can use **var** tags to define four optional variables: **allowallschemes**, **allow2slashes**, **nofragments**, and **schemes**.

The **allowallschemes** and **schemes** variables are used to specify which URL schemes are acceptable when the **url** validation is determining if a URL is valid or not. The **schemes** variable allows you to specify a list of schemes the URL must have in order to pass the validation, as shown here:

```
<field property="websiteUrl" depends="required, url">
   <arg position="0" key="label.sampleForm.websiteUrl"/>
  <var>
    <var-name>schemes</var-name>
    <var-value>http,https</var-value>
  </var>
</field>
```

In this example, only URLs having the "http" or "https" schemes will be valid (e.g., http://domain.com or https://domain.com). A URL with an "ftp" scheme (e.g., ftp://domain.com) would be invalid. The **allowallschemes** variable, shown next, accepts *true* or *false* to specify whether the **url** validation will accept any scheme. This variable defaults to false.

```
<field property="websiteUrl" depends="required, url">
  <arg position="0" key="label.sampleForm.websiteUrl"/>
  <var>
    <var-name>allowallschemes</var-name>
    <var-value>true</var-value>
  </var>
</field>
```

Note that the **allowallschemes** variable overrides the **schemes** variable if both are specified.

The **allow2slashes** variable accepts *true* or *false* to specify whether double slash (/) characters are allowed in the path of URLs (e.g.,http://domain.com/dir1//page.html). This variable defaults to *false*. The **nofragments** variable accepts *true* or *false* to specify whether URL fragments (i.e., anchors) are allowed to be a part of acceptable URLs. This variable defaults to *false* meaning that fragments are allowed (e.g.,http://domain.com/page.html#fragment).

The **url** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.url** entry in the resource bundle file for its error message. The default **errors.url** entry is shown next:

```
errors.url={0} is an invalid URL.
```

The validwhen Validation

The **validwhen** validation is used to determine whether the field being validated is required based on a specified test condition. If the test condition succeeds, the validation passes.

The following snippet illustrates how to configure a **validwhen** validation in a Validator configuration file:

```
<field property="emailConfirm" depends="validwhen">
  <arg position="0" key="label.sampleForm.emailConfirm"/>
```

```
  <var>
    <var-name>test</var-name>
    <var-value>((email == null) or (*this* != null))</var-value>
  </var>
</field>
```

To configure the **validwhen** validation, you specify it in the list of validations for the given field with the **depends** attribute of the **field** tag. You must use an **arg** tag to specify the key for a value in the application resource bundle file or a literal value that will be used to populate the **validwhen** validation's error message. Additionally, you must use the **var** tag to define a **test** variable. The **test** variable is used by the **validwhen** validation to specify a test condition that must pass in order for the validation to succeed.

The test condition specified with the **test** variable is designed to allow for dependent validations to be defined. Dependent validations are validations where one field's validity is based on the value of another field. For example, an email confirmation field may

be required only if a related email field has a value. There is no way to define this interdependent requirement with the standard **required** validation. With the **validwhen** validation, however, you can create a validation dependency from one field to another, as was shown in the previous configuration example.

The **test** variable's test condition is a Boolean expression that must evaluate to true in order for the validation to succeed. Following is the list of rules governing the syntax for creating test conditions:

- Form fields are referenced by their logical name (e.g., name, email, etc.).
- The field for which the validation is being performed is referenced as ***this***.
- **null** is used to indicate null or an empty string.
- Literal string values must be single- or double-quoted.
- Literal integer values can be specified using decimal, hex, or octal formats.
- All comparisons must be enclosed in parentheses.
- A maximum of two comparisons can be made in one condition using **and** or **or**.
- A numeric comparison is performed for conditions that have two numerical values; otherwise, a string comparison is performed.

The **validwhen** validation is set up in the preconfigured **validator-rules.xml** file to use the **errors.required** entry in the resource bundle file for its error message. The default **errors.required** entry is shown next:

```
errors.required={0} is required.
```

Enabling Client-Side Validations

In addition to providing a framework for simplifying server-side form data validations, Validator provides an easy-to-use mechanism for performing client-side validations. Each validation routine defined in the **validator-rules.xml** file optionally specifies JavaScript code that can be run in the browser (client side) to perform the same validations that take place on the server side. When run on the client side, the validations will not allow the form to be submitted until they have all passed. Validations that fail will trigger error dialog windows that specify the fields and the type of validation that failed. Figure 6-1shows a sample error dialog triggered from a failed validation.

Figure 6-1: Validator's client-side JavaScript error dialog

To enable client-side validation, you have to place the HTML Tag Library's **javascript** tag in each JSP for which you want client-side validation performed, as shown here:

```
<html:javascript formName="logonForm"/>
```

The **javascript** tag outputs the JavaScript code necessary to run all of a form's configured validations on the client-side. Note that the **javascript** tag requires that you use the **formName** attribute to specify the name of a **<form>** definition from the **validation.xml** file, as shown here, for which you want validations performed:

```
<form name="logonForm">
  <field property="username" depends="required">
    <arg position="0" key="prompt.username"/>
  </field>
  <field property="password" depends="required">
    <arg position="0" key="prompt.password"/>
  </field>
</form>
```

All the validations that you have specified for the **<form>** definition to run on the server side will be run on the client side.

After adding the **javascript** tag to your page, you must update the HTML Tag Library's **form** tag to have an **onsubmit** attribute. The **onsubmit** attribute is used to specify JavaScript code that is executed when the form is submitted.

In addition to outputting all of the JavaScript code for the validation routines, the **javascript** tag generates a master JavaScript validation method that is used to invoke all of the configured validations for the form. This master method must be called from the **onsubmit** attribute of the **form** tag. Following is an example of how to specify the **onsubmit** attribute for Validator's client-side validations:

```
<html:form action="/Logon" onsubmit="return validateLogonForm(this);">
```

The master validation method generated by the **javascript** tag is named based on concatenating "validate" with the name of the form name specified with the **formName** attribute of the **javascript** tag. For example, a form named "searchForm" will have a generated method named "validateSearchForm".

Note The **method** attribute of the **javascript** tag can be used to specify an alternate method name for the master method that Validator generates. This is useful in scenarios where the auto-generated method name conflicts with other JavaScript method names in your application.

Understanding the Client-Side Validation Process

Validator's client-side validations are performed in a specific order and, as previously mentioned, trigger error dialog windows upon failure. It's important to understand the sequence in which validations are performed in order to understand when error dialogs are triggered. Validator's client-side JavaScript processes validations based on the order they are specified in the **validation.xml** file for a form. Validator proceeds field-by-field processing the validations, but there is one important detail about how the validations are processed. Once a validation is run for a field, for example, the **required**validation, all fields that are to be validated using that validation are run at that time. That is, instead of running a validation each time it is specified in the order it is specified, it is run once for all fields that specify it and then the next validation is processed.

Validator stops processing validations and triggers an error dialog window if any fields being validated by a particular validation fail. This behavior can sometimes be frustrating for users of an application who submit a form that has several validations on it. If multiple validations fail, the user will be prompted only for the first validation that fails. The user can then fix the field(s) with errors and submit the form again. If the validation that failed the first time passes and another validation fails, another error dialog will be displayed. This back-and-forth process of alerting the user of which fields failed validation and then resubmitting the form again for another run of the validations can be cumbersome for the user. The basic problem is that Validator stops processing validations and displays an error dialog for only the first validation that fails. This is the default behavior. However, Validator supports an option for processing all validations, whether they fail or not, and then displaying error dialogs for each validation that failed after all validations have been processed. To enable this option, you have to update the Validator **<plug-in>** definition

in the Struts configuration file, as shown next:

```
<!-- Validator Configuration -->
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property property="pathnames"
                   value="/org/apache/struts/validator/validator-
rules.xml,
                          /WEB-INF/validation.xml"/>
  <set-property property="stopOnFirstError" value="false"/>
</plug-in>
```

The **stopOnFirstError** property has to be set to false in order to have Validator process all validations and then display error dialogs for each failed validation. Each validation that fails will have its own error dialog listing each of the fields that failed that particular validation.

Creating a Common Client-Side Validations JSP

The HTML Tag Library's **javascript** tag outputs the JavaScript code for all validation routines independent of whether they are used or not. For example, even if a form on a page has only two fields and makes use of only the **required** validation for those fields, the **javascript** tag outputs the JavaScript code for the **required** validation and all other validations configured in the **validator-rules.xml** file. This can add a large amount of unnecessary page download overhead to each page making use of client-side validations.

To get around all of the validations being inserted into each page, you can create a common client-side validations JSP that will house the JavaScript for all validations. Each page that makes use of client-side validation will link to the common page, instead of embedding the validations inline. This approach allows the browser to cache the common validations page, thus eliminating the

unnecessary overheard of downloading the validation code for every request of each page. To make use of a common validations JSP, you must first create the common JSP, as shown next:

```
<%@ page contentType="application/x-javascript" language="java" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>


<html:javascript dynamicJavascript="false" staticJavascript="true"/>
```

This page is typically named **staticJavascript.jsp;** however, the name is entirely up to you. Notice that this page uses the **javascript** tag with the**dynamicJavascript** and **staticJavascript** attributes set to false and true, respectively. This instructs the **javascript** tag to output only the JavaScript code for the validation routines and not the code that invokes the routines for the fields of a form. The JavaScript code that invokes the routines is dynamic and must be housed in each of the individual JSPs utilizing client-side validation. Following is the snippet that must be placed in each of the individual JSPs utilizing client-side validation.

```
<html:javascript formName="LogonForm"
        dynamicJavascript="true"
         staticJavascript="false"/>
<script language="JavaScript" src="staticJavascript.jsp"></script>
```

Notice that the **javascript** tag's **dynamicJavascript** and **staticJavascript** attributes are set to true and false respectively in the individual pages; this is the opposite of how they are set in the **staticJavascript.jsp** page. In addition to using the **javascript** tag to emit the dynamic JavaScript code, you must use a **script** tag to link to the common validations JSP.

Creating Custom Validations

Validator comes packaged with several useful validations that will suit most situations; however, your application may require a specific validation that is not provided by the framework. For this situation, Validator provides a simple interface for creating your own custom validations that can easily be plugged into the framework. Following is a list of the steps you need to take to create your own custom validation:

1. Create a new validation method.
2. Add a new validation rule to the **validation.xml** file.
3. Add new validation definitions to the **validation.xml** file.
4. Add messages to the **MessageResources.properties** file.

The following sections walk through each step of the process in detail, enabling you to create a custom validation based on the Social Security Number validation used in the example Mini HR application in Chapter 2. Remember that the Social Security Number validation in Chapter 2 was defined inside of the **SearchForm** Form Bean class. Creating a custom validation for social security numbers enables the validation code to be reused and to be used declaratively instead of being hard-coded in each Form Bean that wants to use it.

Creating a Validation Method

The first step in creating a custom validation is to create a validation method that can be called by the Validator framework. Typically, all of an application's methods for custom validations are grouped into a class of their own, as is the case for the example in this section. However, you can place the method in any class. Your validation method needs to have the following signature:

```
public static boolean validateSsNum(java.lang.Object,
  org.apache.commons.validator.ValidatorAction,
  org.apache.commons.validator.Field,
  org.apache.struts.action.ActionMessages,
  org.apache.commons.validator.Validator,
  javax.servlet.http.HttpServletRequest);
```

Of course, the name of your method will vary, but its arguments should match the types shown in the preceding example. Table 6-2 explains each of the validation method's arguments.

Table 6-2: The Validation Method Arguments

Argument	Description
java.lang.Object	The Form Bean object (downcast to **Object**) contains the field to be validated.
org.apache.commons.validator.ValidatorAction	The **ValidatorAction** object encapsulates the **<validator>** definition from the**validator-rules.xml** file for this validation routine.
org.apache.commons.validator.Field	The **Field** object encapsulates the **<field>** definition from the **validation.xml** file for the field that is currently being validated.
org.apache.struts.action.ActionMessages	The **ActionMessages** object stores validation error messages for the field that is currently being validated.
org.apache.commons.validator.Validator	The **Validator** object encapsulates the current Validator instance and is used to access other field values.
javax.servlet.http.HttpServletRequest	The **HttpServletRequest** object encapsulates the current HTTP request.

Following is the custom validation code for validating social security numbers. Notice that the **validateSsNum( )** method conforms to the proper method signature for custom validations.

```
package com.jamesholmes.minihr;

import javax.servlet.http.HttpServletRequest;
import org.apache.commons.validator.Field;
import org.apache.commons.validator.Validator;
import org.apache.commons.validator.ValidatorAction;
import org.apache.commons.validator.ValidatorUtil;
import org.apache.struts.action.ActionMessages;
import org.apache.struts.validator.Resources;

public class MiniHrValidator
{
```

```
  public static boolean validateSsNum(Object bean,
    ValidatorAction action,
    Field field,
    ActionMessages errors,
    Validator validator,
    HttpServletRequest request)
  {
    String value =
      ValidatorUtil.getValueAsString(bean, field.getProperty());
    if (value == null || value.length() < 11) {
      errors.add(field.getKey(),
        Resources.getActionMessage(validator, request, action, field));
      return false;
    }

    for (int i = 0; i < 11; i++) {
      if (i == 3 || i == 6) {
        if (value.charAt(i) != '-') {
          errors.add(field.getKey(),
            Resources.getActionMessage(validator, request, action,
field));
          return false;
        }
      } else if ("0123456789".indexOf(value.charAt(i)) == -1) {
        errors.add(field.getKey(),
          Resources.getActionMessage(validator, request, action, field));
        return false;
      }
    }

    return true;
  }
}
```

The **validateSsNum( )** method begins by retrieving the value for the field being validated. The value is retrieved by determining the field's name with a call to **field. getProperty( )** and then looking up that field in the Form Bean with a call to **ValidatorUtil. getValueAsString( )**. The **getValueAsString( )**method matches the name of the field with the name of one of the Form Bean fields and then gets that field's value. The rest of the **validateSsNum( )**method performs the actual validation logic. If the validation fails for any reason, an error message will be stored in the **errors ActionMessages** object.

Adding a New Validation Rule

After the custom validation code has been created, a new validation rule needs to be added to the **validation.xml** file. As discussed earlier in this chapter, validation rules "plug" validation methods into the Validator. Once defined in **validation.xml**, as shown here,

the validation rule can be referenced by a **field** tag's **depends** attribute:

```
<validator name="ssNum"
      classname="com.jamesholmes.minihr.MiniHrValidator"
         method="validateSsNum"
   methodParams="java.lang.Object,
                 org.apache.commons.validator.ValidatorAction,
                 org.apache.commons.validator.Field,
                 org.apache.struts.action.ActionMessages,
                 org.apache.commons.validator.Validator,
                 javax.servlet.http.HttpServletRequest"
           msg="errors.ssNum">
  <javascript>
    <![CDATA[
    function validateSsNum(form) {
      var isValid = true;
      var focusField = null;
      var i = 0;
      var fields = new Array();
      var oSsNum = eval('new ' + jcv_retriveFormName(form) + '_ssNum()');

      for (var x in oSsNum) {
        if (!jcv_verifyArrayElement(x, oSsNum[x])) {
          continue;
        }

        var field = form[oSsNum[x][0]];
        if (!jcv_isFieldPresent(field)) {
          continue;
        }

        if ((field.type == 'hidden' ||
             field.type == 'password' ||
             field.type == 'text' ||
             field.type == 'textarea') &&
            (field.value.length > 0))
```

```
        {
          var value = field.value;
          var isRightFormat = true;

          if (value.length != 11) {
            isRightFormat = false;
          }

          for (var n = 0; n < 11; n++) {
            if (n == 3 || n == 6) {
              if (value.substring(n, n+1) != '-') {
                isRightFormat = false;
              }
            } else if ("0123456789".indexOf(
                        value.substring(n, n+1)) == -1) {
              isRightFormat = false;
            }
          }

          if (!isRightFormat) {
            if (i == 0) {
              focusField = field;
            }
            fields[i++] = oSsNum[x][1];
            isValid = false;
          }
        }
        if (fields.length > 0) {
          jcv_handleErrors(fields, focusField);
        }
        return isValid;
      }
      ]]>
    </javascript>
</validator>
```

As you can see, the validation rule applies a name to the validation with the **name** attribute; specifies the class in which the validation method is housed with the **classname** attribute; and specifies the validation method's arguments with the **methodParams** attribute. The **msg** attribute specifies a key

that will be used to look up an error message in the application's resource bundle file (e.g., **MessageResources.properties**) if the validation fails. Note that the name applied with the **name** attribute is the logical name for the rule and will be used to apply the rule to definitions in the **validation.xml** file.

The preceding custom validation rule also defines JavaScript code, inside the opening and closing **<javascript>** elements, which will be used if client-side validation is enabled when using the rule. The JavaScript code simply performs the same validation on the client side as is performed on the server side. If the JavaScript validation fails, it will alert the user and prevent the HTML form from being submitted.

Note that validation rules must be placed in a **<global>** element in the **validation.xml** file, as shown here:

```
<form-validation>
  <global>
    <validator name="ssNum" .../>
    ...
    ...
  </global>
</form-validation>
```

Of course, the order of the **<validator>** elements underneath the **<global>** element is arbitrary, so you can place the new **ssNum** validation rule anywhere.

Adding New Validation Definitions

Once you have defined your custom validation rule in the **validation.xml** file, you can make use of it by referencing its logical name in the **validation.xml**file:

```
<!DOCTYPE form-validation PUBLIC
          "-//Apache Software Foundation//DTD Commons
           Validator Rules Configuration 1.3.0//EN"
          "http://jakarta.apache.org/commons/dtds/validator_1_3_0.dtd">

<form-validation>
  <formset>
    <form name="searchForm">
      <field property="ssNum" depends="required,ssNum">
        <arg position="0" key="prompt.ssNum"/>
      </field>
    </form>
  </formset>
</form-validation>
```

In the preceding validation definition, each of the comma-delimited values of the **field** tag's **depends** attribute corresponds to the logical name of a validation rule defined in the **validation-rules.xml** file. The use of **ssNum** instructs Validator to use the custom Social Security

Number validation. Each time you want to use the new Social Security Number validation, you simply have to add its logical name to the **depends** attribute of a **field** tag.

Adding Messages to the MessageResources.properties File

The final step in creating a custom validation is to add messages to the **MessageResources.properties** file:

```
prompt.ssNum=Social Security Number
errors.ssNum={0} is not a valid Social Security Number
```

Remember that the **errors.ssNum** message key was specified by the **msg** attribute of the **validator** tag for the custom validation rule in the**validation.xml** file. The key's corresponding message will be used if the validation fails. The **prompt.ssNum** message key was specified by the **arg** tag of the validation definition in the **validation.xml** file. Its corresponding message will be used as the parametric replacement for the **errors.ssNum**message's {0} parameter. Thus, if the Social Security Number custom validation fails, the following error message will be generated by substituting the**prompt.ssNum** message for {0} in the **errors.ssNum** message:

```
Social Security Number is not a valid Social Security Number
```

Internationalizing Validations

Similar to other areas of Struts, Validator fully supports internationalization. Remember that internationalization is the process of tailoring content to a specific locale or region. In Validator's case, internationalization means tailoring validation error messages to a specific locale and/or tailoring actual validation routines to a specific locale. This way, the U.S. and French versions of a Web site can each have their own language-specific validation error messages. Similarly, internationalization enables the U.S. and French versions of a Web site to validate entries in monetary fields differently. The U.S. version requires commas to separate dollar values and a period to demarcate cents (i.e., 123,456.78), whereas the French (Euro monetary system) version requires periods to separate dollar amounts and a comma to demarcate cents (i.e., 123.456,78).

Tailoring validation error messages to a specific locale is built into Struts by way of its Resource Bundle mechanism for externalizing application strings, messages, and labels. You simply create a resource bundle file for each locale you want to support. Each locale-specific resource bundle file will have a locale identifier at the end of the filename that denotes which locale it is for, such as **MessageResources_ja.properties** for Japan.

Note For detailed information on internationalizing a Struts application, see Chapter 10.

Thus, when Validator goes to load an error message, it will use the locale object that Struts stores in the session (or from the request if Struts is configured that way) to determine which resource bundle file to load the message from. Remember that each validation rule in the **validator-rules.xml**file specifies a key for a validation error message stored in the application's resource bundle files. This key is the same across each locale's resource bundle file, thus allowing Validator to load the appropriate message based on only a locale and a key.

Tailoring validation routines to specific locales is similar to tailoring error messages; you have to define validation definitions in the **validation.xml** file for each locale. The **validation.xml** file contains a **form-validation** tag that contains one or more **formset** tags, which in turn contain one or more **form**tags, and so on:

```
<!DOCTYPE form-validation PUBLIC
```

```
          "-//Apache Software Foundation//DTD Commons
            Validator Rules Configuration 1.3.0//EN"
          "http://jakarta.apache.org/commons/dtds/validator_1_3_0.dtd">

<form-validation>
  <formset>
    <form name="auctionForm">
      <field property="bid" depends="mask">
        <var>
          <var-name>mask</var-name>
          <var-value>^\d{1,3}(,?\d{3})*\.?(\d{1,2})?$</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

The **form-set** tag takes optional attributes, **country**, **language**, and **variant**, to tailor its nested forms to a specific locale. In the preceding example, all locales use the same currency validation to validate the "bid" property, because none of the **country**, **language**, or **variant** attributes was specified for the **form-set** tag.

In order to have a generic currency validation for all users of the Web site and a Euro-specific currency validation for users from France, you'd create an additional **form-set** definition specifically for the French users, as shown here:

```
<!DOCTYPE form-validation PUBLIC
          "-//Apache Software Foundation//DTD Commons
            Validator Rules Configuration 1.3.0//EN"
          "http://jakarta.apache.org/commons/dtds/validator_1_3_0.dtd">

<form-validation>
  <formset>
    <form name="auctionForm">
      <field property="bid" depends="required,mask">
        <var>
          <var-name>mask</var-name>
          <var-value>^\d{1,3}(,?\d{3})*\.?(\d{1,2})?$</var-value>
        </var>
      </field>
    </form>
  </formset>
```

```
  <formset country="fr">
    <form name="auctionForm">
      <field property="bid" depends="required,mask">
        <var>
          <var-name>mask</var-name>
          <var-value>^\d{1,3}(\.?\d{3})*,?(\d{1,2})?$</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

In this listing, the second **<formset>** definition specifies a **country** attribute set to "fr". That instructs Validator to use the enclosed validation definitions for users with a French locale. Notice that the bid validation in the second **<formset>** definition is different from the first definition, as the period and comma are transposed in the mask value. Thus, the first **<formset>** definition validates that bids are in U.S. currency format and the second definition validates that bids are in Euro currency format.

A powerful feature of using internationalized **<formset>** definitions is that you can define only the validations that are locale-specific, and all other validations are taken from the default **<formset>** definition.

Adding Validator to the Mini HR Application

Now that you've seen the benefits of using the Validator framework and how it works, you are ready to revisit the Mini HR application and replace the hard-coded validation logic with Validator. Following is the list of steps involved in adding the Validator to the Mini HR application:

1. Change **SearchForm** to extend **ValidatorForm**.
2. Create a **validation.xml** file.
3. Add the Validator plugin to the **struts-config.xml** file.
4. Add Validation error messages to the **MessageResources.properties** file.
5. Recompile, repackage, and run the updated application.

The following sections walk through each step of the process in detail.

Change SearchForm to Extend ValidatorForm

The first step in converting the Mini HR application to use Validator is to change **SearchForm** to extend Validator's **ValidatorForm** class instead of extending Struts' basic **ActionForm** class. Recall that **ValidatorForm** extends **ActionForm** and provides an implementation for its **reset()** and**validate( )** methods that hook into the Validator framework; thus, those methods should be removed from the **SearchForm** class. Additionally, the**isValidSsNum( )** method should be removed from the **SearchForm** class because its functionality is being replaced by Validator as well.

Following is the updated **SearchForm.java** file:

```
package com.jamesholmes.minihr;


import java.util.List;
```

```
import org.apache.struts.validator.ValidatorForm;

public class SearchForm extends ValidatorForm
{
  private String name = null;
  private String ssNum = null;
  private List results = null;

  public void setName(String name) {
    this.name = name;
  }
  public String getName() {
    return name;
  }

  public void setSsNum(String ssNum) {
    this.ssNum = ssNum;
  }

  public String getSsNum() {
    return ssNum;
  }

  public void setResults(List results) {
    this.results = results;
  }

  public List getResults() {
    return results;
  }
}
```

Notice that this file no longer has the **reset( )**, **validate( )**, and **validateSsNum( )** methods and that the class been updated to extend **ValidatorForm**.

Create a validation.xml File

After removing the hard-coded validation logic from **SearchForm**, you must create a **validation.xml** file. This file will inform Validator which validations from the **validator-rules.xml** file should be applied to **SearchForm**. Following is a basic **validation.xml** file that validates that social security numbers have the proper format if entered:

```
<!DOCTYPE form-validation PUBLIC
```

```
          "-//Apache Software Foundation//DTD Commons
            Validator Rules Configuration 1.3.0//EN"
          "http://jakarta.apache.org/commons/dtds/validator_1_3_0.dtd">

<form-validation>
  <formset>
    <form name="searchForm">
      <field property="ssNum" depends="mask">
        <arg position="0" key="label.search.ssNum"/>
        <var>
          <var-name>mask</var-name>
          <var-value>^\d{3}-\d{2}-\d{4}$</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

Notice that this file does not contain any validation definitions to ensure that either a name or a social security number was entered, the way the original hard-coded logic did. This is because such logic is complicated to implement with Validator and thus should be implemented using Struts' basic validation mechanism.

Add the Validator Plugin to the struts-config.xml File

After setting up Validator's configuration file, the following snippet must be added to the Struts configuration file to cause Struts to load the Validator plugin:

```
<!-- Validator Configuration -->
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property property="pathnames"
                value="/org/apache/struts/validator/validator-rules.xml,
                       /WEB-INF/validation.xml"/>
</plug-in>
```

Notice that each of the configuration files is specified with the **set-property** tag. The following snippet lists the updated Struts configuration file for Mini HR in its entirety:

```
<!DOCTYPE struts-config PUBLIC
  "-//Apache Software Foundation//DTD Struts Configuration 1.3//EN"
  "http://struts.apache.org/dtds/struts-config_1_3.dtd">

<struts-config>
```

```
  <!-- Form Beans Configuration -->
  <form-beans>
    <form-bean name="searchForm"
               type="com.jamesholmes.minihr.SearchForm"/>
  </form-beans>

  <!-- Global Forwards Configuration -->
  <global-forwards>
    <forward name="search" path="/search.jsp"/>
  </global-forwards>

  <!-- Action Mappings Configuration -->
  <action-mappings>
    <action path="/search"
            type="com.jamesholmes.minihr.SearchAction"
            name="searchForm"
            scope="request"
            validate="true"
            input="/search.jsp">
    </action>
  </action-mappings>

  <!-- Message Resources Configuration -->
  <message-resources
    parameter="com.jamesholmes.minihr.MessageResources"/>

  <!-- Validator Configuration -->
  <plug-in className="org.apache.struts.validator.ValidatorPlugIn">
    <set-property property="pathnames"
                  value="/org/apache/struts/validator/validator-rules.xml,
                         /WEB-INF/validation.xml"/>
  </plug-in>

</struts-config>
```

Add Validation Error Messages to the MessageResources.properties File

Recall from earlier in this chapter that each validation routine defined in the **validator-rules. xml** file declares a key for an error message in Struts' resource bundle file: **MessageResources.properties**. At run time, Validator uses the keys to look up error messages to return when validations fail.

Because you are using the **mask** validation defined in the **validator-rules. xml** file, you must add the following error message for its declared key to the**MessageResources.properties** file:

```
errors.invalid={0} is not valid
```

The following code shows the updated **MessageResources.properties** file in its entirety:

```
# Label Resources
label.search.name=Name
label.search.ssNum=Social Security Number

# Error Resources
error.search.criteria.missing=Search Criteria Missing
error.search.ssNum.invalid=Invalid Social Security Number
errors.header=<font color="red"><cTypeface:Bold>Validation
Error(s)</b></font><ul>
errors.footer=</ul><hr width="100%" size="1" noshade="true">
errors.prefix=<li>
errors.suffix=</li>

errors.invalid={0} is not valid
```

Compile, Package, and Run the Updated Application

Because you removed the **reset( )**, **validate( )**, and **validateSsNum( )** methods from **SearchForm** and changed it to extend **ValidatorForm** instead of**ActionForm**, you need to recompile and repackage the Mini HR application before you run it. Assuming that you've made modifications to the original Mini HR application and it was set up in the **c:\java\ MiniHR** directory (as described in Chapter 2), you can run the **build.bat** batch file or the **build.xml**Ant script file to recompile the application.

After recompiling Mini HR, you need to repackage it using the following command line:

```
jar cf MiniHR.war *
```

This command should also be run from the directory where you have set up the Mini HR application (e.g., **c:\java\MiniHR**).

Similar to the way you ran Mini HR the first time, you now need to place the new **MiniHR.war** file that you just created into Tomcat's **webapps** directory, delete the **webapps/MiniHR** directory, and start Tomcat. As before, to access the Mini HR application, point your browser to**http://localhost:8080/MiniHR/**. Once you have the updated Mini HR running, try entering valid and invalid social security numbers. As you will see, they are now verified using the new Validator code.